Programming Language Selection for Optimization Algorithms*

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

January 22, 2010
Technical Report: CA-TR-20100122-1

Abstract

The Clever Algorithms project aims to describe a large number of optimization algorithms from the field of Artificial Intelligence in a complete, consistent, and centralized manner. Source code examples will be included in the algorithm descriptions, and as such a suitable programming language must be selected for use throughout the project. A methodology for selecting a suitable programming language is proposed, involving an assessment of commonly used languages by their popularity of use with a typical optimization algorithm. A set of requirements for language selection are proposed, and a practical algorithm comparison strategy is adopted where a typical optimization algorithm is implemented and compared between for candidate programming languages. Results show the four candidate languages (Python, Perl, Ruby, and Lua) are generally very similar in function and syntax, although the Ruby programming language is potentially more expressive and concise. Some clearly cited biases in the comparison are highlighted.

Keywords: `Clever, Algorithms, Programming, Language, Selection, Optimization, Python, Perl, Ruby, Lua, Methodology`

1 Introduction

The Clever Algorithms project aims to describe a large number of optimization algorithms from the fields of Computational Intelligence, Natural Computation, and Metaheuristics in a complete, consistent, and centralized manner [1]. The standardized description of algorithms in the project requires an example implementation of each technique in a tutorial and potentially a code listing format [2]. A practical example is required for each algorithm description to show how the abstract procedures can be implemented as a concrete and executable computation. This report addresses the problem of which language to use for describing algorithm implementation examples, and recommends a specific language to be used.

Section 2 considers the problem of language selection focusing on the types of languages that could be used and examples, as well as a consideration of attributes that might be important when selecting a language. Section 3 proposes a methodology for evaluating and comparing languages specifying an algorithm to use in the comparison, a set of languages to be considered, and the specific properties that will be measured and compared. Four languages are considered: Python in Section 4, Perl in Section 5, Ruby in Section 6 and Lua in Section 7. Finally, the

results of the comparison are reviewed in Section 9 highlighting areas for improvement in the language selection methodology and in the execution of the comparison. The Ruby Programing Language was selected as the programming language to be used for code examples throughout the Clever Algorithms project.

2 Language Selection

This section considers the importance of the programming language used in implementation examples in general, and looks at some languages and common attributes that may be relevant.

2.1 Languages

Programming languages provide a means for a human to specify a computation, procedure, or application in a form and structure that can be understood and executed by a computer. A introduction to programming languages is beyond the scope of this report, rather this section provides a brief overview of some programming language features, nomenclature and taxonomy suitable to make general decisions about and comparisons between popular programming languages.

An important concern of a programming language is the organizational paradigm that includes the methodology, abstractions, style, and structures. Some common example paradigms include:

- *Logical*: Involves the use of mathematical logic (such as propositional or predicate logic) for computer programming. Programs are comprised of declarative logical statements (propositions) that may be used to describe a rule-based reasoning or knowledge representation system. An example is Prolog.

- *Functional*: Focuses on the execution of mathematical functions (from lambda calculus) and avoids state and mutable data. The difference between a mathematical function and an imperative function is that the latter may have side effects (modify state beyond the scope of the function), whereas the former may not. Some examples include LISP, Scheme, Erlang, and Haskell.

- *Procedural*: Also referred to as imperative programming that is concerned with organizing code into procedures (also called subroutines or functions). Procedures are defined in terms of a series expressions, some of which may in themselves be calls to procedures. Some common examples include: Pascal and C.

- *Object-Oriented*: Models problems using objects that are comprised of data and procedures (also called methods). Objects may have an abstract definition (a class) and may be instantiated within a program. Objects can interact with each other by passing messages between each other, such as one object invoking a method on another object. Some examples include C++, Java and C#.

- *Other*: There are many programming language paradigms not listed here, not limited to reflective, declarative, agent-oriented, aspect-oriented, event-driven, and meta programming. Most modern programming languages are not constrained to a single programming paradigm (called multi-paradigm), such as Java (imperative, object-oriented, generic, reflective) and Ruby (imperative, object-oriented, aspect-oriented, reflective, functional).

Another consideration of a programming language is whether or not programs are compiled or interpreted (dynamic). Compiled languages are those which use a compiler to transform a higher-level language to a lower-level language. The compilation process can introduce various

forms of static structure and type checking that may improve the quality of software produced, although the added steps between source code and execution may affect the speed and momentum of development. Some examples of compiled languages include C and C++ that compile to machine code, and Java and C# that compile to bytecode and intermediate language respectively (incidentally, the latter two examples represent compilation to an intermediate language that is interpreted by a virtual machine).

Dynamic languages are high-level languages that execute at runtime rather than requiring compilation to machine code before execution. They can be more productive for small specific tasks given the typically high-levels of abstraction and language features provided (such as dynamic typing and meta-programming), although they can be relatively slower during exception compared to equivalent complied languages[1].

Additional aspects of computer programming languages that may be relevant include types: such as whether a language is typed or untyped, and if typed, whether types are static or dynamic.

2.2 Languages Attributes

The programming language attributes that are important for describing algorithms in a book or website medium are different from those concerns for selecting a language for implementing an application, although there may be some overlap. Traditional language selection concerns such as *Static or Dynamically Typed*, *Compiled or Interpreted*, and *Single or Multi-paradigms* may be resolved by higher-level concerns such as popularity and commonality in the field and appropriateness in the industry and by communication concerns such as compactness and readability.

2.2.1 Practitioner

The practitioner is concerned with the usability of an example, either directly in a language that can be exploited in a current project, or indirectly in a language or format that can easily be adapted for a current project. Practitioners are most likely using the common languages in the industry for scientific and application computer programming, such as C, C++, C#, Java, and potentially dynamic languages such as Python and Ruby. Paradigms used by practitioners are mostly likely Object-Oriented programming (OO) and Procedural programming. The following questions motivate language selection from the perspective of the practitioner:

- What languages are most commonly used by practitioners in industry and/or academia?
- What language paradigms are most commonly used in algorithm implementations?
- Which languages provide the most appropriate functionalities in their standard library (such as random number generation, vector manipulation, etc.)?

2.2.2 Communication

The communication perspective of implementation examples is concerned less with practical concerns than usability, such as understandably, readability, and accessibility. Given a varied readership of amateurs, scientists, and developers, procedural programming examples would be the most easy to understand (modules of steps of expressions) followed closely by object-oriented programming. System programming languages (such as C) are likely to be concise, although scripting programming languages such as Lua, Ruby, and Python are likely to provide compact source code listings. The following questions motivate language selection from the perspective of implementation communication:

[1] Just-in-time compilation can improve the dynamic performance of a program by compiling to a native format at runtime, and is offered by some interpreted languages.

- Given the varied background of the readership, is the average reader likely to easily understand the example?
- How large (number of printed lines or pages) is an an average implementation example take up?
- How readable (perhaps aesthetics such as balance of composition) is a given implementation example?

3 Methodology

This section presents a methodology for comparing programming languages for presenting optimization (and similar) heuristic methods from the fields of Computational Intelligence, Natural Computation, and Metaheuristics in a book and/or website medium. The methodology is empirical, requiring a selection of a representative algorithm, the selection of some candidate programming languages for the selected algorithm to be programmed in, and the identification of some requirements and guidelines for each implementation such that they can be compared.

3.1 Algorithm

The algorithm used for comparison must be representative of the algorithms that require implementation examples. A first-draft of all algorithms to be described in the Clever Algorithms project has been defined [2]. In that definition, popularity of each technique was measured against popular search engines. Algorithm popularity provides a reasonable basis for representation. The top five popular algorithms were: genetic algorithm, simulated annealing, random search, genetic programming, and tabu search.

The genetic algorithm provides a suitable representative algorithm in that it is not only popular, but also involves a number of non-trivial operators (compared to random or tabu search), although is relatively compact (compared to genetic programming that involves management of tree structures). Algorithm 1 provides an abstract representation of a standardized genetic algorithm in pseudo code. Simulated annealing also provides a suitable representative algorithm, and may be used in followup and/or confirmatory studies.

3.2 Implementations

Implementing an algorithm in a range of different programming languages is time consuming. Heuristics are required to reduce the set of possible languages down to a manageable number that can be compared. This section considers algorithm popularity as a heuristic in a similar manner to the way in which algorithms were selected for description in the Clever Algorithms project. A listing of common and popular languages was prepared and a script written to measure the number of search results for each language across a range of search domains: Google web, book, and scholar (in Section 3.2.1). The requirements for a selected language are described based on generalizations from language classes (from Section 2) and from the needs of the readership (in Section 3.2.2). Finally, a subset of popular algorithms were selected for comparison (in Section 3.2.3).

3.2.1 Language Popularity

Table 1 lists the results of measuring the popularity of 24 popular programming languages. The measures were taken from the Google web, book, and scholar search services on January 20th 2010 for the phrase '*"NAME programming language" AND "genetic algorithm"*' (where 'NAME' is replaced with each specific language name). The table of results show algorithm

Algorithm 1: Pseudo Code for the Genetic Algorithm.

```
Input: popSize, numBits, p_crossover, p_mutation
Output: Best
Population ← InitializeRandomBitstrings(popSize, numBits);
foreach p_i ∈ Population do
    Fitness(p_i);
end
Best ← Sort(Population).Last;
while ¬StopCondition() do
    Population_children ← 0;
    while Population_children.size()≠popSize do
        parent_1 ← TournamentSelection(Population, boutSize);
        parent_2 ← TournamentSelection(Population, boutSize);
        child_1, child_2 ← Crossover(p_2, p_2, p_crossover);
        mutant_1 ← Mutate(child_1, p_mutation);
        mutant_2 ← Mutate(child_2, p_mutation);
        Population_children ← mutant_1;
        Population_children ← mutant_2;
    end
    foreach p_i ∈ Population do
        Fitness(p_i);
    end
    Best ← Sort(Population).Last;
    Replace(Population, Population_children);
end
return Best;
```

name, a raw approximation of the number of results from each service and the score as the sum of the normalized approximate measures. Results are ordered by score, descending.

Language	Google Web	Google Book	Google Scholar	Score
c	940	91	801	3.0
java	735	39	553	1.899
c++	673	42	9	1.187
lisp	163	51	139	0.902
matlab	166	24	138	0.607
python	238	7	88	0.435
erlang	375	0	0	0.395
pascal	182	11	30	0.347
ada	161	10	16	0.296
fortran	62	9	39	0.208
scheme	105	3	13	0.155
perl	100	1	35	0.155
c#	91	1	30	0.139
ruby	108	1	4	0.125
visual basic	35	3	46	0.121
lua	59	0	1	0.058
scala	31	0	1	0.028
php	21	0	7	0.025
ocaml	27	0	2	0.025
objective-c	14	1	2	0.022
mathematica	6	1	4	0.016
smalltalk	13	0	6	0.015
haskell	12	0	1	0.008
javascript	6	0	0	0.0

Table 1: Algorithm programming language listing measures.

3.2.2 Language Requirements

This section lists the requirements for selecting a language.

- *The language shall allow programming in the procedural paradigm.* A procedural representation is expected to provide the most transferrable instantiation of an algorithm. Many languages support the procedural paradigm and procedural code examples can be easily ported to popular paradigms such as object-oriented and functional.
- *The language shall be interpreted.* The compilation of a program is an additional step to the reader in executing a provided code example. In some languages this step is minimal, in others it may require an understanding of the intermediate representation and linking. An interpreted program can be executed by an interpret in a single step, useful for modification and exploration by the reader.
- *The language shall be available and accessible at no monetary cost.* Some languages require a proprietary framework or application to compile or interpret the source code (such as .NET), some of which must be purchased (such as Mathematica and Matlab). There may be exceptions and open source equivalents, although the main distribution for the language shall require no monetary cost to acquire and use.

3.2.3 Candidate Languages

The following are the languages selected for comparison based on the popularity heuristics and the language selection requirements:

1. The Python Programming Language
2. The Perl Programming Language
3. The Ruby Programming Language
4. The Lua Programming Language

The selected languages are listed in order of popularity from Table 1. The languages are interpreted (dynamic), available at no cost, and support the procedural programming paradigm. The Lua programming language is technically compiled, although the compilation process occurs at execution time and therefore remains an interpreted language (from a user perspective). The following list specifies guidelines for the algorithm implementations across the four languages.

- Procedural (imperative) programming paradigm (functions and data structures, no objects).
- Promote readability over fancy language tricks (such as Regex and one-liners).
- Strive toward < 80 characters per line (code listings wrap automatically at whitespace break points).
- Use the same general procedures (algorithm decomposition) across language implementations, for consistency.
- Target < 100 lines per implementation example.
- Avoid the use of magic numbers, prefer the use of constants.
- Avoid compressing flow control structures to one or fewer lines (if-else,while,for).
- No comments in implementations.

The algorithm implementations across the languages should use the same number of procedures with the same functionality, and the the same constants. The algorithm shall use a bit-string (string data type) implementation, one-point crossover, point mutation, tournament selection and be applied to a 64-bit instance of the One-Max problem.

4 Python

Algorithm Listing 1 provides an example of the Genetic Algorithm implemented in the Python Programming Language. Tested against the Python interpreter version 2.6.1.

```
1  import random
2
3  NUM_GENERATIONS = 100
4  NUM_BOUTS = 3
5  POP_SIZE = 100
6  NUM_BITS = 64
7  P_CROSSOVER = 0.98
8  P_MUTATION = 1.0/NUM_BITS
9  HALF = 0.5
10
11 def onemax(bitstring):
12   sum = 0
```

```
  for c in bitstring:
    if(c=='1'):
      sum += 1
  return sum

def tournament(population):
  best = None
  for i in range(NUM_BOUTS):
    other = population[random.randint(0, len(population)-1)]
    if best==None or other['fitness']>best['fitness']:
      best = other
  return best

def mutation(bitstring):
  string = ''
  for c in bitstring:
    if random.random()<P_MUTATION:
      if c=='1':
        string += '0'
      else:
        string += '1'
    else:
      string += c
  return string

def crossover(parent1, parent2):
  if random.random() < P_CROSSOVER:
    cut = random.randint(1, NUM_BITS-1)
    return parent1['bitstring'][0:cut]+parent2['bitstring'][cut:NUM_BITS],
        parent2['bitstring'][0:cut]+parent1['bitstring'][cut:NUM_BITS]
  return ''+parent1['bitstring'], ''+parent2['bitstring']

def random_bitstring():
  s = ''
  for x in range(NUM_BITS):
    if random.random() < HALF:
      s += '0'
    else:
      s += '1'
  return s

def evolve():
  population = []
  for x in range(POP_SIZE):
    population.append({'bitstring':random_bitstring(), 'fitness':0})
  for candidate in population:
    candidate['fitness'] = onemax(candidate['bitstring'])
  population.sort(lambda x, y: x['fitness']-y['fitness'])
  gen, best = 0, population[POP_SIZE-1]
  while best['fitness']!=NUM_BITS and gen<NUM_GENERATIONS:
    children = []
    while len(children) < POP_SIZE:
      s1, s2 = crossover(tournament(population), tournament(population))
      children.append({'bitstring':mutation(s1), 'fitness':0})
      if len(children) < POP_SIZE:
        children.append({'bitstring':mutation(s2), 'fitness':0})
    for candidate in children:
      candidate['fitness'] = onemax(candidate['bitstring'])
    children.sort(lambda x, y: x['fitness']-y['fitness'])
    if children[POP_SIZE-1]['fitness'] > best['fitness']:
      best = children[POP_SIZE-1]
    population = children
    gen += 1
```

```
    print " > gen %d, best: %d, %s" % (gen, best['fitness'], best['bitstring'])
  return best

best = evolve()
print "done! Solution: f=%d, s=%s" % (best['fitness'], best['bitstring'])
```

Listing 1: Genetic Algorithm in the Python Programming Language

5 Perl

Algorithm Listing 2 provides an example of the Genetic Algorithm implemented in the Perl Programming Language. Tested against the Perl interpreter version 5.10.0.

```
use constant NUM_GENERATIONS => 100;
use constant NUM_BOUTS => 3;
use constant POP_SIZE => 100;
use constant NUM_BITS => 64;
use constant P_CROSSOVER => 0.98;
use constant P_MUTATION => (1.0/NUM_BITS);
use constant HALF => 0.5;

sub onemax {
  my $bitstring = $_[0];
  my $sum = 0;
  $sum += $1 while $bitstring =~ s/^(.)//;
    return $sum;
}

sub mutation {
  my $bitstring = $_[0];
  $bitstring =~ s/(.)/rand() < P_MUTATION ? $1^1 : $1/ge;
  return $bitstring;
}

sub crossover {
  my ($parent1, $parent2) = ($_[0], $_[1]);
  if(rand() < P_CROSSOVER) {
    my $cut = int(rand(NUM_BITS-2)) + 1;
    return (substr($parent1, 0, $cut).substr($parent2, $cut), substr($parent2, 0,
        $cut).substr($parent1, $cut));
  }
  return (".$parent1, ".$parent2);
}

sub random_bitstring {
  my $string = "-" x NUM_BITS;
  $string =~ s/(.)/rand() < HALF ? 0 : 1/ge;
  return $string;
}

sub tournament {
  my @population = @{$_[0]};
  my $best = '';
  for my $p (0..(NUM_BOUTS-1)) {
    $i = int(rand(@population));
    if($best eq '' or $population[$i]{fitness} > ${$best}{fitness}){
      $best = $population[$i];
    }
  }
  return $best;
}

```

```
sub evolve {
  my @population;
  for my $p (0..(POP_SIZE-1)) {
    push @population, {bitstring=>random_bitstring(), fitness=>0};
  }
  for $candidate (@population) {
    $candidate->{fitness} = onemax($candidate->{bitstring});
  }
  my @sorted = sort{$b->{fitness} <=> $a->{fitness}} @population;
  my $gen = 0;
  my $best = $sorted[0];
  while(${$best}{fitness}<NUM_BITS and $gen<NUM_GENERATIONS) {
    my @children;
    while(@children < POP_SIZE) {
      $p1 = tournament(\@population);
      $p2 = tournament(\@population);
      my ($c1, $c2) = crossover(${$p1}{bitstring}, ${$p2}{bitstring});
      push @children, {bitstring=>mutation($c1), fitness=>0};
      if(@children < POP_SIZE) {
        push @children, {bitstring=>mutation($c2), fitness=>0};
      }
    }
    for $candidate (@children) {
      $candidate->{fitness} = onemax($candidate->{bitstring});
    }
    @sorted = sort{$b->{fitness} <=> $a->{fitness}} @children;
    if($sorted[0]{fitness}>${$best}{fitness}) {
      $best = $sorted[0];
    }
    @population = @children;
    $gen = $gen + 1;
    print " > gen $gen, best: ${$best}{fitness}, ${$best}{bitstring}\n";
  }

  return $best;
}

$best = evolve();
print "done! Solution: best: ${$best}{fitness}, ${$best}{bitstring}\n";
```

Listing 2: Genetic Algorithm in the Perl Programming Language

6 Ruby

Algorithm Listing 3 provides an example of the Genetic Algorithm implemented in the Ruby Programming Language. Tested against the ruby interpreter version 1.8.7 and 1.9.1.

```
NUM_GENERATIONS = 100
NUM_BOUTS = 3
POP_SIZE = 100
NUM_BITS = 64
P_CROSSOVER = 0.98
P_MUTATION = 1.0/NUM_BITS
HALF = 0.5

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

def tournament(population)
```

```ruby
  best = nil
  NUM_BOUTS.times do
    other = population[rand(population.size)]
    best = other if best.nil? or other[:fitness]>best[:fitness]
  end
  return best
end

def mutation(source)
  string = ""
  source.each_char do |bit|
    if rand<P_MUTATION
      string << ((bit=='1') ? "0" : "1")
    else
      string << "#{bit}"
    end
  end
  return string
end

def crossover(parent1, parent2)
  if rand < P_CROSSOVER
    cut = rand(NUM_BITS-2) + 1
    return parent1[:bitstring][0...cut]+parent2[:bitstring][cut...NUM_BITS],
        parent2[:bitstring][0...cut]+parent1[:bitstring][cut...NUM_BITS]
  end
  return ""+parent1[:bitstring], ""+parent2[:bitstring]
end

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

def evolve
  population = Array.new(POP_SIZE) do |i|
    {:bitstring=>random_bitstring, :fitness=>0}
  end
  population.each{|c| c[:fitness] = onemax(c[:bitstring])}
  gen, best = 0, population.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
  while best[:fitness]!=NUM_BITS and gen<NUM_GENERATIONS
    children = []
    while children.size < POP_SIZE
      s1, s2 = crossover(tournament(population), tournament(population))
      children << {:bitstring=>mutation(s1), :fitness=>0}
      children << {:bitstring=>mutation(s2), :fitness=>0} if children.size < POP_SIZE
    end
    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
    gen += 1
    puts " > gen #{gen}, best: #{best[:fitness]}, #{best[:bitstring]}"
  end
  return best
end

best = evolve()
puts "done! Solution: f=#{best[:fitness]}, s=#{best[:bitstring]}"
```

Listing 3: Genetic Algorithm in the Ruby Programming Language

7 Lua

Algorithm Listing 4 provides an example of the Genetic Algorithm implemented in the Lua Programming Language. Tested against the Lua interpreter version 5.1.4.

```
NUM_GENERATIONS = 100
NUM_BOUTS = 3
POP_SIZE = 100
NUM_BITS = 64
P_CROSSOVER = 0.98
P_MUTATION = 1.0/NUM_BITS
HALF = 0.5

function onemax(bitstring)
  local sum = 0
  for i=1, bitstring:len() do
    if(bitstring:sub(i,i) == "1") then
      sum = sum+1
    end
  end
  return sum
end

function tournament(population)
  local best = nil
  for i=1, NUM_BOUTS do
    local other = population[math.random(#population)]
    if(best==nil or other.fitness > best.fitness) then
      best = other
    end
  end
  return best
end

function mutation(bitstring)
  local string = ""
  for i=1, bitstring:len() do
    local c = bitstring:sub(i,i)
    if math.random() < P_MUTATION then
      if c == "0" then
        string = string.."1"
      else
        string = string.."0"
      end
    else
      string = string..c
    end
  end
  return string
end

function crossover(parent1, parent2)
  if math.random() < P_CROSSOVER then
    local cut = math.random(NUM_BITS-2) + 2
    return parent1.bitstring:sub(1,cut-1)..parent2.bitstring:sub(cut,NUM_BITS),
        parent2.bitstring:sub(1,cut-1)..parent1.bitstring:sub(cut,NUM_BITS)
  end
  return ""..parent1.bitstring, ""..parent2.bitstring
end

function random_bitstring()
  local s = ""
  for i=1, NUM_BITS do
```

```
    if math.random() < HALF then
      s = s.."0"
    else
      s = s.."1"
    end
  end
  return s
end

function evolve()
  local population = {}
  for i=1, POP_SIZE do
    table.insert(population, {bitstring=random_bitstring(),fitness=0})
  end
  for i,candidate in ipairs(population) do
    candidate.fitness = onemax(candidate.bitstring)
  end
  table.sort(population, function(a,b) return a.fitness<b.fitness end)
    local gen, best = 0, population[POP_SIZE]
  while best.fitness<NUM_BITS and gen<NUM_GENERATIONS do
    local children = {}
    while #children < POP_SIZE do
      local s1, s2 = crossover(tournament(population), tournament(population))
      table.insert(children, {bitstring=mutation(s1),fitness=0})
      if #children < POP_SIZE then
        table.insert(children, {bitstring=mutation(s2),fitness=0})
      end
    end
    for i,candidate in ipairs(children) do
      candidate.fitness = onemax(candidate.bitstring)
    end
    table.sort(children, function(a,b) return a.fitness<b.fitness end)
    if(children[POP_SIZE].fitness > best.fitness) then
      best = children[POP_SIZE]
    end
    population = children
    io.write(" > gen "..gen..", best: "..best.bitstring.."\n")
    gen = gen + 1
  end
  return best
end

best = evolve()
io.write("done! Solution:f="..best.fitness..", s="..best.bitstring.."\n")
```

Listing 4: Genetic Algorithm in the Lua Programming Language

8 Results

Three measures were collected over the for algorithm-language implementations, including: *Total lines* that assess the absolute readable length of an example, *Total Chars* that provides some information about the expressiveness of the language, and *Avg Cars/Line* that provides an indication of the density of the source code. Smaller numbers are better for all three measures, although the guidelines from section 3.2.3 apply. The results are provided in Table 2 and show the basic analytics of the source files for each of the four algorithm implementations.

9 Analysis

This section analyses both the methodology and the results of the language comparison.

Algorithm	Total Lines	Total Chars	Avg Chars/Line
Python	79	1959	25
Perl	87	2104	24
Ruby	72	1764	25
Lua	101	2202	22

Table 2: Algorithm programming language source code measures.

9.1 Methodology

Language comparison is notoriously difficult because of the syntactic characteristics and functional capabilities can differ so wildly from language-to-language. The proposed methodology considered algorithm popularity as a starting point for selecting languages, and candidate selection using such imposed constraints as procedural, interpreted, and free. These constraints were based on heuristics, although were not supported with direct evidence. It is possible, even likely that popular compiled languages used in industry such as C#, Java, and C++ would be the most relevant for the target audience for the Clever Algorithms Project. The benefits of the imposed constraints are that algorithm examples presented in a procedural, interpreted, and free language may easily be ported to such languages.

9.2 Implementation

It can be difficult to implement the same computation across different languages, and the four examples in this report demonstrate this fact. Three of the four languages are syntactically and functionally very similar (Python, Ruby, and Lua), with Perl representing marked differences. As a result, the computational differences between Perl and the other language are quite noticeable, specifically in the smaller utility functions.

The results in Table 2 show that the Ruby example was implemented in fewer lines and characters than the other programming languages, with similar per-line character density. This may highlight the expressive power of the language compared to the other language, and/or the biases of the author in wielding the respective languages. Either way, compactness and conciseness under the constraints of readability and accessibility are the primary desirable properties for algorithm implementation examples in the Clever Algorithms project. **The Ruby Programming Language has been selected for all code examples in the Clever Algorithms project**.

It may also be apparent to those familiar with the respective languages used in the comparison that the Ruby implementation is slightly more considered than the other examples. This was because the Ruby example was implemented first and ported to the other languages. This fact may have introduced a bias into the comparison, beyond the inherent experience biases of the author of the examples.

The biases introduced by using a single author with variable experience across the selected languages may be addressed by submitting the code examples to peer-review and refinement by language experts. This process was initially explored with the Perl code example where a language expert was consulted. For such a comparison to be meaningful, such code reviews are required. In addition to language experience, the algorithm may also be restructured to make it easier to implement across the selected languages, or for a specific language, within the readability constraints. This avenue may be explored by testing alternative functional decompositions for the algorithm.

Features like meta-programming and postfix loops and conditions (such as: `a=b if condition`) provide readability improvements over classical functional programming and are considered an advantage of using modern interpreted languages like those candidates considered. Although

implementation guidelines were provided and generally adopted, there are examples of where the code examples are not necessarily readable. Two examples include the use of regular expressions in the Perl example (lines 12 and 18), and the use of a one-liner in the Ruby example on line 45. A readability concern that is obvious across all four implementations is the relative long-length and high-density of the `evolve()` function. These examples highlight the need for iteration and revision of code examples in order to identify and refine specific functionality in code examples throughout the Clever Algorithms project.

10 Acknowledgments

Thank-you to Paul Maisano for his review and suggestions for the Perl programming language code example used in this work.

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.