Clonal Selection Algorithm*

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

May 15, 2010
Technical Report: CA-TR-20100515-1

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 Clonal Selection Algorithm using the standardized template.

Keywords: `Clever, Algorithms, Description, Optimization, Clonal, Selection`

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 Clonal Selection Algorithm using the standardized template.

2 Name

Clonal Selection Algorithm, CSA, CLONALG

3 Taxonomy

The Clonal Selection Algorithm (CLONALG) belongs to the field of Artificial Immune Systems. It is related to other Clonal Selection algorithms such as the Artificial Immune Recognition System (AIRS), the B-Cell Algorithm (BCA), and the Multi-objective Immune System Algorithm (MISA). There are numerious extensions to CLONALG including tweaks such as the CLONALG1 and CLONALG2 approaches, a version for classification called CLONCLAS, and an adaptive version called Adaptive Clonal Selection (ACS).

*© Copyright 2010 Jason Brownlee. Some Rights Reserved. This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 2.5 Australia License.

4 Inspiration

The Clonal Selection algorithm is inspired by the Clonal Selection theory of acquired immunity. The clonal selection theory credited to Burnet was proposed to account for the behavior and capabilities of antibodies in the acquired immune system [5, 6]. Inspired itself by the principles of Darwinian natural selection theory of evolution, the theory proposes that antigens select-for lymphocytes (both B and T-cells). When a lymphocyte is selected and binds to an antigenic determinant, the cell proliferates making many thousands more copies of itself and differentiates into different cell types (plasma and memory cells). Plasma cells have a short lifespan and produce vast quantities of antibody molecules, whereas memory cells live for an extended period in the host anticipating future recognition of the same determinant. The important feature of the theory is that when a cell is selected and proliferates, it is subjected to small copying errors (changes to the genome called somatic hypermutation) that change the shape of the expressed receptors and subsequent determinant recognition capabilities of both the antibodies bound to the lymphocytes cells surface, and the antibodies that plasma cells produce.

5 Metaphor

The theory suggests that starting with an initial repertoire of general immune cells, the system is able to change itself (the compositions and densities of cells and their receptors) in response to experience with the environment. Through a blind process of selection and accumulated variation on the large scale of many billions of cells, the acquired immune system is capable of acquiring the necessary information to protect the host organism from the specific pathogenic dangers of the environment. It also suggests that the system must anticipate (guess) at the pathogen to which it will be exposed, and requires exposure to pathogen that may harm the host before it can acquire the necessary information to provide a defense.

6 Strategy

The information processing principles of the clonal selection theory describe a general learning strategy. This strategy involves a population of adaptive information units (each representing a problem-solution or component) subjected to a competitive processes for selection, which together with the resultant duplication and variation ultimately improves the adaptive fit the information units to their environment.

7 Procedure

Algorithm 1 provides a pseudo-code listing of the Clonal Selection Algorithm (CLONALG) for minimizing a cost function. The general CLONALG model involves the selection of antibodies (candidate solutions) based on affinity either by matching against an antigen pattern or via evaluation of a pattern by a cost function. Selected antibodies are subjected to cloning proportional to affinity, and the hypermutation of clones inversely-proportional to clone affinity. The resultant clonal-set competes with the existent antibody population for membership in the next generation. In addition, low-affinity population members are replaced by randomly generated antibodies. The pattern recognition variation of the algorithm includes the maintenance of a memory solution set which in its entirety represents a solution to the problem. A binary-encoding scheme is employed for the binary-pattern recognition and continuous function optimization examples, and an integer permutation scheme is employed for the Traveling Salesman Problem (TSP).

Algorithm 1: Pseudo Code for the Clonal Selection Algorithm (CLONALG).

```
Input: $Population_{size}$, $Selection_{size}$, $Problem_{size}$, $RandomCells_{num}$, $Clone_{rate}$,
       $Mutation_{rate}$
Output: Population
1  Population ← CreateRandomCells($Population_{size}$, $Problem_{size}$);
2  while ¬StopCondition() do
3      foreach $p_i$ ∈ Population do
4          Affinity($p_i$);
5      end
6      $Population_{select}$ ← Select(Population, $Selection_{size}$);
7      $Population_{clones}$ ← 0;
8      foreach $p_i$ ∈ $Population_{select}$ do
9          $Population_{clones}$ ← Clone($p_i$, $Clone_{rate}$);
10     end
11     foreach $p_i$ ∈ $Population_{clones}$ do
12         Hypermutate($p_i$, $Mutation_{rate}$);
13         Affinity($p_i$);
14     end
15     Population ← Select(Population, $Population_{clones}$, $Population_{size}$);
16     $Population_{rand}$ ← CreateRandomCells($RandomCells_{num}$);
17     Replace(Population, $Population_{rand}$);
18 end
19 return Population;
```

8 Heuristics

- The CLONALG was designed as a general machine learning approach and has been applied to pattern recognition, function optimization, and combinatorial optimization problem domains.
- Binary string representations are used and decoded to a representation suitable for a specific problem domain.
- The number of clones created for each selected member is calculated as a function of the repertoire size $N_c = round(\beta \cdot N)$, where β is the user parameter $Clone_{rate}$.
- A rank-based affinity-proportionate function is used to determine the number of clones created for selected members of the population for pattern recognition problem instances.
- The number of random antibodies inserted each iteration is typically very low (1-2).
- Point mutations (bit-flips) are used in the hypermutation operation.
- The function $exp(-\rho \cdot f)$ is used to determine the probability of individual component mutation for a given candidate solution, where f is the candidates affinity (normalized maximizing cost value), and ρ is the user parameter $Mutation_{rate}$.

9 Code Listing

Listing 1 provides an example of the Clonal Selection Algorithm (CLONALG) implemented in the Ruby Programming Language. The demonstration problem is an instance of a continuous function optimization that seeks $min f(x)$ where $f = \sum_{i=1}^{n} x_i^2$, $-5.0 \leq x_i \leq 5.0$ and $n = 3$. The optimal solution for this basin function is $(v_0, \ldots, v_{n-1}) = 0.0$. The algorithm is implemented as described by de Castro and Von Zuben for function optimization [10].

```ruby
BITS_PER_PARAM = 16

def objective_function(vector)
  return vector.inject(0.0) {|sum, x| sum + (x**2.0)}
end

def decode(bitstring, search_space)
  vector = []
  search_space.each_with_index do |bounds, i|
    off, sum, j = i*BITS_PER_PARAM, 0.0, 0
    bitstring[off...(off+BITS_PER_PARAM)].reverse.each_char do |c|
      sum += ((c=='1') ? 1.0 : 0.0) * (2.0 ** j.to_f)
      j += 1
    end
    min, max = bounds
    vector << min + ((max-min)/((2.0**BITS_PER_PARAM.to_f)-1.0)) * sum
  end
  return vector
end

def evaluate(pop, search_space)
  pop.each do |p|
    p[:vector] = decode(p[:bitstring], search_space)
    p[:cost] = objective_function(p[:vector])
  end
end

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

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

def affinity_proportionate_mutation(cost, mutate_rate)
  cost = cost * -1.0 if cost<0
  return Math.exp(-2.5 * cost)
end

def num_clones(pop_size, clone_factor)
  return (pop_size * clone_factor).to_i
end

def calculate_affinity(pop)
  max = pop.max{|x,y| x[:cost]<=>y[:cost]}
  min = pop.min{|x,y| x[:cost]<=>y[:cost]}
  range = max[:cost]-min[:cost]
  if range == 0
    pop.each {|p| p[:affinity] = 1.0}
  else
    pop.each {|p| p[:affinity] = 1.0-(p[:cost]-min[:cost]/range)}
  end
end

def clone_and_hypermutate(pop, clone_factor, mutate_factor)
```

```
  clones = []
  num_clones = num_clones(pop.size, clone_factor)
  calculate_affinity(pop)
  pop.each do |antibody|
    p_mutation = affinity_proportionate_mutation(antibody[:affinity], mutate_factor)
    num_clones.times do
      clone = {}
      clone[:bitstring] = ""+antibody[:bitstring]
      point_mutation(clone[:bitstring], p_mutation)
      clones << clone
    end
  end
  return clones
end

def greedy_merge(pop, clones)
  union = pop + clones
  union.sort!{|x,y| x[:cost]<=>y[:cost]}
  return union[0...pop.size]
end

def random_insertion(search_space, pop, problem_size, num_rand)
  return pop if num_rand == 0
  rands = Array.new(num_rand) do |i|
    {:bitstring=>random_bitstring(problem_size*BITS_PER_PARAM)}
  end
  evaluate(rands, search_space)
  return greedy_merge(pop, rands)
end

def search(problem_size, search_space, max_gens, pop_size, clone_factor, mutate_factor,
    num_rand)
  pop = Array.new(pop_size) do |i|
    {:bitstring=>random_bitstring(problem_size*BITS_PER_PARAM)}
  end
  evaluate(pop, search_space)
  gen, best = 0, pop.min{|x,y| x[:cost]<=>y[:cost]}
  max_gens.times do |gen|
    clones = clone_and_hypermutate(pop, clone_factor, mutate_factor)
    evaluate(clones, search_space)
    pop = greedy_merge(pop, clones)
    pop = random_insertion(search_space, pop, problem_size, num_rand)
    best = (pop + [best]).min{|x,y| x[:cost]<=>y[:cost]}
    puts " > gen #{gen+1}, f=#{best[:cost]}, a=#{best[:affinity]} s=#{best[:vector].inspect}"
  end
  return best
end

problem_size = 3
max_gens = 200
pop_size = 100
clone_factor = 0.1
mutate_factor = 2.5
num_rand = 2
search_space = Array.new(problem_size) {|i| [-5, +5]}

best = search(problem_size, search_space, max_gens, pop_size, clone_factor, mutate_factor,
    num_rand)
puts "done! Solution: f=#{best[:cost]}, s=#{best[:vector].inspect}"
```

Listing 1: Clonal Selection Algorithm (CLONALG) in the Ruby Programming Language

10 References

10.1 Primary Sources

Hidden at the back of a technical report on the applications of Artificial Immune Systems de Castro and Von Zuben [11] proposed the Clonal Selection Algorithm (CSA) as a computational realization of the clonal selection principle for pattern matching and optimization. The algorithm was later published [9], and investigated where it was renamed to CLONALG (CLONal selection ALGorithm) [10].

10.2 Learn More

Watkins, et al. proposed to exploit the *inherent distributedness* of the CLONALG and proposed a parallel version of the pattern recognition version of the algorithm [13]. White and Garret also investigated the pattern recognition version of CLONALG and generalized the approach for the task of binary pattern classification renaming it to Clonal Classification (CLONCLAS) where their approach was compared to a number of simple Hamming distance based heuristics [14]. In an attempt to address concerns of algorithm efficiency, parameterization, and representation selection for continuous function optimization Garrett proposed an updated version of CLONALG called Adaptive Clonal Selection (ACS) [12]. In their book, de Castro and Timmis provide a detailed treatment of CLONALG including a description of the approach (starting page 79) and a step through of the algorithm (starting page 99) [7]. Cutello and Nicosia provide a study of the clonal selection principle and algorithms inspired by the theory [8]. Brownlee provides a review of Clonal Selection algorithms providing a taxonomy, algorithm reviews, and a broader bibliography [1].

11 Conclusions

This report described the Clonal Selection Algorithm (CLONALG). Elements of this report were drawn from some of the author's previous works including a review of Clonal Selection Algorithms [1] and his dissertation work [2].

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. Clonal selection algorithms. Technical Report 070209A, Complex Intelligent Systems Laboratory (CIS), Centre for Information Technology Research (CITR), Faculty of Information and Communication Technologies (ICT), Swinburne University of Technology, Feb 2007.

[2] Jason Brownlee. *Clonal Selection as an Inspiration for Adaptive and Distributed Information Processing.* PhD thesis, Complex Intelligent Systems Laboratory, Faculty of Information and Communication Technologies, Swinburne University of Technology, 2008.

[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] F. M. Burnet. A modification of jernes theory of antibody production using the concept of clonal selection. *Australian Journal of Science*, 20:67–69, 1957.

[6] F. M. Burnet. *The clonal selection theory of acquired immunity.* Vanderbilt University Press, 1959.

[7] Leandro N. De Castro and Jonathan Timmis. *Artificial immune systems: a new computational intelligence approach.* Springer, 2002.

[8] Vincenzo Cutello and Giuseppe Nicosia. *Recent Developments in Biologically Inspired Computing*, chapter Chapter VI. The Clonal Selection Principle for In Silico and In Vivo Computing, pages 104–146. Idea Group Publishing, 2005.

[9] Leandro N. de Castro and Fernando J. Von Zuben. The clonal selection algorithm with engineering applications. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '00), Workshop on Artificial Immune Systems and Their Applications*, pages 36–37, 2000.

[10] Leandro N. de Castro and Fernando J. Von Zuben. Learning and optimization using the clonal selection principle. *IEEE Transactions on Evolutionary Computation*, 6:239–251, 2002.

[11] Leandro N. de Castro and Fernando Jose Von Zuben. Artificial immune systems - part i: Basic theory and applications. Technical Report TR DCA 01/99, Department of Computer Engineering and Industrial Automation, School of Electrical and Computer Engineering, State University of Campinas, Brazil, 1999.

[12] Simon M. Garrett. Parameter-free, adaptive clonal selection. In *Congress on Evolutionary Computing (CEC 2004)*, pages 1052–1058, 2004.

[13] Andrew Watkins, Xintong Bi, and Amit Phadke. Parallelizing an immune-inspired algorithm for efficient pattern recognition. In *Intelligent Engineering Systems through Artificial Neural Networks: Smart Engineering System Design: Neural Networks*, pages 225–230, 2003.

[14] Jennifer White and Simon M. Garrett. Improved pattern recognition with artificial clonal selection? In *Proceedings Artificial Immune Systems: Second International Conference, ICARIS 2003*, pages 181–193, 2003.