Learning Classifier System*

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

April 11, 2010
Technical Report: CA-TR-20100411-1

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 Learning Classifier System algorithm using the standardized template.

Keywords: `Clever, Algorithms, Description, Optimization, Learning, Classifier, System`

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 Learning Classifier System algorithm using the standardized template.

2 Name

Learning Classifier System, LCS

3 Taxonomy

The Learning Classifier System algorithm is both an instance of an Evolutionary Algorithm from the field of Evolutionary Computation and an instance of a Reinforcement Learning algorithm from Machine Learning. The Learning Classifier System is a theoretical system with a number of implementations. Two streams of classifier are the Pittsburgh-style that seeks to optimize whole classifier, and the Michigan-style that optimize responsive rulesets. The Michigan-style Learning Classifier is the most common and is comprised of two versions: the ZCS (zeroth-level classifier system) and the XCS (accuracy-based classifier system).

4 Strategy

The objective of the Learning Classifier System algorithm is to optimize payoff based on exposure to stimuli from a problem-specific environment. This is achieved by managing credit assignment for those rules that prove useful and searching for new rules and new variations on existing rules using an evolutionary process.

5 Procedure

The actors of the system include detectors, messages, effectors, feedback, and classifiers. Detectors are used by the system to perceive the state of the environment. Messages are the discrete information packets passed from the detectors into the system. The system performs information processing on messages, and messages may directly result in actions in the environment. Effectors control the actions of the system on and within the environment. In addition to the system actively perceiving via its detections, it may also receive directed feedback from the environment (payoff). Classifiers are condition-action rules that provides a filter for messages. If a message satisfies the conditional part of the classifier, the action of the classier triggers. Rules act as message processors. Messages are defined at a fixed length using a binary alphabet. A classifier is defined as a binary string with a ternary alphabet of $1, 0, \#$, where the $\#$ represents do not care (matching both a 1 or 0).

The processing loop for the Learning Classifier system is as follows: i) Messages from the environment are placed on the message list. ii) The conditions of each classifier are checked to see if they are satisfied by at least one message in the message list. iii) All classifiers that are satisfied participate in a competition, those that win post their action to the message list. iv) All messages directed to the effectors are executed (causing actions in the environment). v) All messages on the message list from the previous cycle are deleted (messages persist for a single cycle). The algorithm may be described in terms of the main processing loop and two sub-algorithms: a reinforcement learning algorithm such as the bucket brigade algorithm or Q-learning, and a genetic algorithm for optimization of the system. Algorithm 1 provides a pseudo-code listing of the high-level processing loop of the Learning Classifier System, specifically the XCS as described by Butz and Wilson [6].

6 Heuristics

The majority of the heuristics in this section are specific to the XCS Learning Classifier System as described by Butz and Wilson [6].

- Learning Classifier Systems are suited for problems with the following characteristics: perpetually novel events with large amounts of noise, continual, and real-time requirements for action, implicitly or inexactly defined goals, and sparse payoff or reinforcement obtainable only through long sequences of tasks.
- The learning rate β for a classifiers expected payoff, error and fitness are typically in the range $\in [0.1, 0.2]$.
- The frequency of running the genetic algorithm θ_{GA} should be in the range $\in [25, 50]$.
- The discount factor used in multi-step programs γ are typically in the around 0.71.
- The minimum error for whereby classifiers are considered to have equal accuracy ϵ_0 are typically 10% of the maximum reward.
- The probability of crossover in the genetic algorithm χ are typically in the range $\in [0.5, 1.0]$.

Algorithm 1: Pseudo Code for the Learning Classifier System algorithm.

```
Input: env
Output: Population
env ← InitializeEnvironment(env);
Population ← InitializePopulation();
ActionSet_{t-1} ← 0;
Input_{t-1} ← 0;
Reward_{t-1} ← 0;
while ¬StopCondition() do
    Input_t ← env;
    Matchset ← GenerateMatchSet(Population, Input_t);
    Prediction ← GeneratePrediction(Matchset);
    Action ← SelectionAction(Prediction);
    ActionSet_t ← GenerateActionSet(Action, Matchset);
    Reward_t ← ExecuteAction(Action, env);
    if ActionSet_{t-1} ≠ 0 then
        Payoff_t ← CalculatePayoff(Reward_{t-1}, Prediction);
        PerformLearning(ActionSet_{t-1}, Payoff_t, Population);
        RunGeneticAlgorithm(ActionSet_{t-1}, Input_{t-1}, Population);
    end
    if LastStepOfTask(env, Action) then
        Payoff_t ← Reward_t;
        PerformLearning(ActionSet_t, Payoff_t, Population);
        RunGeneticAlgorithm(ActionSet_t, Input_t, Population);
        ActionSet_{t-1} ← 0;
    else
        ActionSet_{t-1} ← ActionSet_t;
        Input_{t-1} ← Input_t;
        Reward_{t-1} ← Reward_t;
    end
end
```

- The probability of mutating a single position in a classifier in the genetic algorithm μ is typically in the range $\in [0.01, 0.05]$.
- The experience threshold during classifier deletion θ_{del} is typically about 20.
- The experience threshold for a classifier during subsumption θ_{sub} is typically around 20.
- The initial values for a classifiers expected payoff p_1, error ϵ_1, and fitness f_1 are typically small and close to zero.
- The probability of selecting a random action for the purposes of exploration p_{exp} is typically close to 0.5.
- The minimum number of different actions that must be specified in a match set θ_{mna} is usually the total number of possible actions in the environment for the input.
- Subsumption should be used on problem domains that are known contain well defined rules for mapping inputs to outputs.

7 Code Listing

Listing 1 provides an example of the Learning Classifier System algorithm implemented in the Ruby Programming Language. The problem is an instance of a Boolean multiplexer called the 6-multiplexer. It can be described as a classification problem, where each of the 2^6 patterns of bits is associated with a boolean class $\{1, 0\}$. For this problem instance, the first two bits may be decoded as an address into the remaining four bits that specify the class (for example in 100010, '10' decode to the index of '2' in the remaining 4 bits making the class '1'). In propositional logic this problem instance may be described as $F = (\neg x_0)(\neg x_1)x_2 + (\neg x_0)x_1x_3 + x_0(\neg x_1)x_4 + x_0x_1x_5$. The algorithm is an instance of XCS based on the description provided by Butz and Wilson [6] with the parameters based on the application of XCS to Boolean multiplexer problems by Wilson [16, 17]. The population is grown as needed, and subsumption which would be appropriate for the Boolean multiplexer problem was not used for brevity. The multiplexer problem is a single step problem, so the complexities of delayed payoff are not required. A number of parameters were hard coded to recommended values, specifically: $\alpha = 0.1, \upsilon = 5, \delta = 0.1$ and $P_{\#} = \frac{1}{3}$.

```ruby
def new_classifier(condition, action, gen)
  other = {}
  other[:condition], other[:action], other[:lasttime] = condition, action, gen
  other[:prediction], other[:error], other[:fitness] = 0.00001, 0.00001, 0.00001
  other[:experience], other[:setsize], other[:num] = 0.0, 1.0, 1.0
  return other
end

def copy_classifier(parent)
  copy = {}
  parent.keys.each {|k| copy[k] = (parent[k].kind_of? String) ? ""+parent[k] : parent[k]}
  copy[:num] = 1
  copy[:experience] = 0.0
  return copy
end

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

def neg(bit)
  return (bit==1) ? 0 : 1
end

def target_function(s)
  ints = Array.new(s.length){|i| s[i].chr.to_i}
  x0,x1,x2,x3,x4,x5 = ints
  return neg(x0)*neg(x1)*x2 + neg(x0)*x1*x3 + x0*neg(x1)*x4 + x0*x1*x5
end

def calculate_deletion_vote(classifier, pop, del_thresh)
  vote = classifier[:setsize] * classifier[:num]
  avg_fit = pop.inject(0.0){|s,c| s+c[:fitness]}/pop.inject(0.0){|s,c| s+c[:num]}
  derated = classifier[:fitness] / classifier[:num]
  if classifier[:experience] > del_thresh and derated < 0.1 * avg_fit
    vote *= avg_fit / derated
  end
  return vote
end

def delete_from_pop(pop, pop_size, del_thresh)
  total = pop.inject(0) {|s,c| s+c[:num]}
  return if total < pop_size
  pop.each {|c| c[:dvote] = calculate_deletion_vote(c, pop, del_thresh)}
  vote_sum = pop.inject(0.0) {|s,c| s+c[:dvote]}
```

```
  point = rand() * vote_sum
  vote_sum, index = 0.0, 0
  pop.each_with_index do |c,i|
    vote_sum += c[:dvote]
    if vote_sum > point
      index = i
      break
    end
  end
  if pop[index][:num] > 1
    pop[index][:num] -= 1
  else
    pop.delete_at(index)
  end
end

def generate_random_classifier(input, actions, gen)
  condition = ""
  input.each_char {|s| condition << ((rand<1.0/3.0) ? '#' : s)}
  action = actions[rand(actions.length)]
  return new_classifier(condition, action, gen)
end

def does_match(input, condition)
  i = 0
  condition.each_char do |c|
    return false if c!='#' and c!=input[i].chr
    i += 1
  end
  return true
end

def get_actions(pop)
  return [] if pop.empty?
  set = {}
  pop.each do |classifier|
    key = classifier[:action]
    set[key] = 0 if set[key].nil?
    set[key] += 1
  end
  return set.keys
end

def generate_match_set(input, pop, all_actions, gen, pop_size, del_thresh)
  match_set = pop.select{|c| does_match(input, c[:condition])}
  actions = get_actions(match_set)
  while actions.length < all_actions.length do
    remaining = all_actions - actions
    classifier = generate_random_classifier(input, remaining, gen)
    pop << classifier
    match_set << classifier
    delete_from_pop(pop, pop_size, del_thresh)
    actions << classifier[:action]
  end
  return match_set
end

def generate_prediction(input, match_set)
  prediction = {}
  match_set.each do |classifier|
    key = classifier[:action]
    prediction[key] = {:sum=>0.0,:count=>0.0,:weight=>0.0} if prediction[key].nil?
    prediction[key][:sum] += classifier[:prediction]*classifier[:fitness]
```

```
    prediction[key][:count] += classifier[:fitness]
  end
  prediction.keys.each do |key|
    prediction[key][:weight]=prediction[key][:sum]/prediction[key][:count]
  end
  return prediction
end

def select_action(prediction_array, p_explore)
  keys = prediction_array.keys
  return true, keys[rand(keys.length)] if rand() < p_explore
  keys.sort!{|x,y| prediction_array[y][:weight]<=>prediction_array[x][:weight]}
  return false, keys.first
end

def update_set(action_set, payoff, l_rate)
  action_set.each do |c|
    c[:experience] += 1.0
    pdiff = payoff - c[:prediction]
    c[:prediction] += (c[:experience]<1.0/l_rate) ? pdiff/c[:experience] : l_rate*pdiff
    diff = pdiff.abs - c[:error]
    c[:error] += (c[:experience]<1.0/l_rate) ? diff/c[:experience] : l_rate*diff
    sum = action_set.inject(0.0) {|s,other| s+other[:num]-c[:setsize]}
    c[:setsize] += (c[:experience]<1.0/l_rate) ? sum/c[:experience] : l_rate*sum
  end
end

def update_fitness(action_set, min_error, l_rate)
  sum = 0.0
  accuracy = Array.new(action_set.length)
  action_set.each_with_index do |c,i|
    accuracy[i] = (c[:error]<min_error) ? 1.0 : 0.1*(c[:error]/min_error)**-5.0
    sum += accuracy[i] * c[:num]
  end
  action_set.each_with_index do |c,i|
    c[:fitness] += l_rate * (accuracy[i] * c[:num] / sum - c[:fitness])
  end
end

def can_run_genetic_algorithm(action_set, gen, ga_freq)
  total = action_set.inject(0.0) {|s,c| s+c[:lasttime]*c[:num]}
  sum = action_set.inject(0.0) {|s,c| s+c[:num]}
  if gen - (total/sum) > ga_freq
    return true
  end
  return false
end

def select_parent(pop)
  sum = pop.inject(0.0) {|s,c| s+c[:fitness]}
  point = rand() * sum
  sum = 0
  pop.each do |c|
    sum += c[:fitness]
    return c if sum > point
  end
end

def mutation(classifier, p_mut, action_set, input)
  classifier[:condition].length.times do |i|
    if rand() < p_mut
      if classifier[:condition][i].chr == '#'
        classifier[:condition][i] = input[i]
```

```
      else
        classifier[:condition][i] = '#'
      end
    end
  end
  if rand() < p_mut
    new_action = nil
    begin
      new_action = action_set[rand(action_set.length)]
    end until new_action != classifier[:action]
    classifier[:action] = new_action
  end
end

def uniform_crossover(string1, string2)
  rs = ""
  string1.length.times do |i|
    rs << ((rand()<0.5) ? string1[i] : string2[i])
  end
  return rs
end

def insert_in_pop(classifier, pop)
  pop.each do |c|
    if classifier[:condition]==c[:condition] and classifier[:action]==c[:action]
      c[:num] += 1
      return
    end
  end
  pop << classifier
end

def crossover(c1, c2, p1, p2)
  c1[:condition] = uniform_crossover(p1[:condition], p2[:condition])
  c2[:condition] = uniform_crossover(p1[:condition], p2[:condition])
  c1[:prediction] = (p1[:prediction]+p2[:prediction])/2.0
  c1[:error] = 0.25*(p1[:error]+p2[:error])/2.0
  c1[:fitness] = 0.1*(p1[:fitness]+p2[:fitness])/2.0
  c2[:prediction] = c1[:prediction]
  c2[:error] = c1[:error]
  c2[:fitness] = c1[:fitness]
end

def run_genetic_algorithm(all_actions, pop, action_set, input, gen, p_cross, p_mut, pop_size,
    del_thresh)
  p1, p2 = select_parent(action_set), select_parent(action_set)
  c1, c2 = copy_classifier(p1), copy_classifier(p2)
  crossover(c1, c2, p1, p2) if rand() < p_cross
  [c1,c2].each do |c|
    mutation(c, p_mut, all_actions, input)
    insert_in_pop(c, pop)
    delete_from_pop(pop, pop_size, del_thresh)
  end
end

def search(length, pop_size, max_gens, all_actions, p_explore, l_rate, min_error, ga_freq,
    p_cross, p_mut, del_thresh)
  pop, abs = [], 0
  max_gens.times do |gen|
    input = generate_problem_string(length)
    match_set = generate_match_set(input, pop, all_actions, gen, pop_size, del_thresh)
    prediction_array = generate_prediction(input, match_set)
    explore, action = select_action(prediction_array, p_explore)
```

```ruby
    action_set = match_set.select{|c| c[:action]==action}
    expected = target_function(input)
    payoff = ((expected-action.to_i)==0) ? 300.0 : 1.0
    abs += (expected - action.to_i).abs.to_f
    update_set(action_set, payoff, l_rate)
    update_fitness(action_set, min_error, l_rate)
    if can_run_genetic_algorithm(action_set, gen, ga_freq)
      action_set.each {|c| c[:lasttime] = gen}
      run_genetic_algorithm(all_actions, pop, action_set, input, gen, p_cross, p_mut, pop_size, del_thresh)
    end
    if (gen+1).modulo(50)==0
      puts " >gen=#{gen+1} classifiers=#{pop.size}, error=#{abs.to_i}/50 (#{(abs/50*100)}%)"
      abs = 0
    end
  end
  return pop
end

max_gens, length, pop_size = 5000, 6, 150
all_actions = ['0', '1']
l_rate, min_error = 0.2, 0.01
p_explore, p_cross, p_mut = 0.10, 0.80, 0.04
ga_freq, del_thresh = 50, 20

pop = search(length, pop_size, max_gens, all_actions, p_explore, l_rate, min_error, ga_freq, p_cross, p_mut, del_thresh)
puts "done! Solution: classifiers=#{pop.size}"
```

Listing 1: Learning Classifier System algorithm in the Ruby Programming Language

8 References

8.1 Primary Sources

Early ideas on the theory of Learning Classifier Systems were proposed by Holland [7, 10], culminating in a standardized presentation a few years later [8]. A number of implementations of the theoretical system were investigated, although a taxonomy of the two main streams was proposed by De Jong [12]: 1) Pittsburgh-style proposed by Smith [14, 13] and 2) Holland-style or Michigan-style Learning classifiers that are further comprised of the Zeroth-level classifier (ZCS) [15] and the accuracy-based classifier (XCS) [16].

8.2 Learn More

Booker, Goldberg, and Holland provide a classical introduction to Learning Classifier Systems including an overview of the state of the field and the algorithm in detail [1]. Wilson and Goldberg also provide a classical introduction and review of the approach, although take a more critical stance [18]. Holmes, et al. provide a contemporary review of the field focusing both on the approach and application areas to which the approach has been demonstrated successfully [11]. Lanzi, Stolzmann, and Wilson provide a seminal book in the field as a collection of papers covering the basics, advanced topics, and demonstration applications. A particular highlight from this book is the first section that provides a concise description of Learning Classifier Systems by many leaders and major contributors to the field [9], providing rare insight. Another paper from this book by Lanzi and Riolo provides a detailed review of the development of the approach as it matured throughout the 1990s. Bull and Kovacs a second book introductory book to the field focusing on the theory of the approach and its practical application [5].

9 Conclusions

This report described the Learning Classifier System as a machine learning technique using both reinforcement learning and genetic algorithms. The content for this report based based on a previous work on the Learning Classifier System by this author [2].

10 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] L. B. Booker, D. E. Goldberg, and J. H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40:235–282, 1989.

[2] Jason Brownlee. Learning classifier systems. Technical Report 070514A, Complex Intelligent Systems Laboratory, Centre for Information Technology Research, Faculty of Information and Communication Technologies, Swinburne University of Technology, May 2007.

[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] Larry Bull and Tim Kovacs. *Foundations of learning classifier systems*. Springer, 2005.

[6] M. V. Butz and S. W. Wilson. An algorithmic description of xcs. *Journal of Soft Computing*, 6(3–4):144–153, 2002.

[7] J. H. Holland. *Progress in Theoretical Biology IV*, chapter Adaptation, pages 263–293. Academic Press, 1976.

[8] J. H. Holland. Adaptive algorithms for discovering and using general patterns in growing knowledge-bases. *International Journal of Policy Analysis and Information Systems*, 4:217–240, 1980.

[9] John H. Holland, Lashon B. Booker, Marco Colombetti, Marco Dorigo, David E. Goldberg, Stephanie Forrest, Rick L. Riolo, Robert E. Smith, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson. *Learning classifier systems: from foundations to applications*, chapter What is a learning classifier system?, pages 3–32. Springer, 2000.

[10] John H. Holland and Judith S. Reitman. Cognitive systems based on adaptive algorithms. *ACM SIGART Bulletin*, 63:49, 1977.

[11] J. H. Holmes, P. L. Lanzi, W. Stolzmann, and S. W. Wilson. Learning classifier systems: New models, successful applications. *Information Processing Letters*, 82:23–30, 2002.

[12] Kenneth De Jong. Learning with genetic algorithms: An overview. *Machine Learning*, 3:121–138, 1988.

[13] S. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proceedings 8th International Joint Conference on Artificial Intelligence*, pages 422–425, 1983.

[14] Stephen Frederick Smith. *A learning system based on genetic adaptive algorithms.* PhD thesis, Department of Computer Science, University of Pittsburgh, 1980.

[15] S. W. Wilson. Zcs: A zeroth level classifier systems. *Evolutionary Computation*, 2:1–18, 1994.

[16] S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3:149–175, 1995.

[17] S. W. Wilson. Generalization in the xcs classifier system. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 665–674. Morgan Kaufmann, 1998.

[18] Stewart W. Wilson and David E. Goldberg. A critical review of classifier systems. In *Proceedings of the third international conference on Genetic algorithms*, pages 244–255, 1989.