Strength Pareto Evolutionary Algorithm*

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

May 8, 2010
Technical Report: CA-TR-20100508-1

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 Strength Pareto Evolutionary Algorithm using the standardized template.

Keywords: `Clever, Algorithms, Description, Optimization, Strength, Pareto, Evolutionary, Algorithm`

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 [1]. 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 [2]. This report describes the Strength Pareto Evolutionary Algorithm using the standardized template.

2 Name

Strength Pareto Evolutionary Algorithm, SPEA, SPEA2

3 Taxonomy

Strength Pareto Evolutionary Algorithm is a Multiple Objective Optimization (MOO) algorithm and an Evolutionary Algorithm (EA) from the field of Evolutionary Computation (EC). It belongs to the field of Evolutionary Multiple Objective (EMO) algorithms. Strength Pareto Evolutionary Algorithm is an extension of the Genetic Algorithm for multiple objective optimization problems. It is related to sibling Evolutionary Algorithms such as Non-dominated Sorting Genetic Algorithm (NSGA), Vector-Evaluated Genetic Algorithm (VEGA), and Pareto

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

Archived Evolution Strategy (PAES). There are two versions of SPEA, the original SPEA algorithm and the extension SPEA2. Additional extensions include SPEA+ and iSPEA.

4 Strategy

The objective of the algorithm is to locate and and maintain a front of non-dominated Pareto optimal solutions. This is achieved by using an evolutionary process (with surrogate procedures for genetic recombination and mutation) to explore the search space, and a selection process that uses a combination of the degree to which a candidate solution is dominated (strength) and an estimation of density of the Pareto front as an assigned fitness. An archive of the Pareto front is maintained separate from the population of candidate solutions used in the evolutionary process, providing a form of elitism.

5 Procedure

Algorithm 1 provides a pseudo-code listing of the Strength Pareto Evolutionary Algorithm 2 (SPEA2) for minimizing a cost function. The `CalculateRawFitness` function calculates the raw fitness as the sum of the strength values of the solutions that dominate a given candidate, where strength is the number of solutions that a give solution dominate. The `CandidateDensity` function estimates the density of an area of the Pareto front as $\frac{1.0}{\sigma^k+2}$ where σ^k is the Euclidean distance of the objective values between a given solution the kth nearest neighbor of the solution, and k is the square root of the size of the population and archive combined. The `PopulateWithRemainingBest` function iteratively fills the archive with the remaining candidate solutions in order of fitness. The `RemoveMostSimilar` function truncates the archive population removing those members with the smallest σ^k values as calculated against the archive. The `SelectParents` function selects parents from a population using a Genetic Algorithm selection method such as binary tournament selection. The `CrossoverAndMutation` function performs the crossover and mutation genetic operators from the Genetic Algorithm.

6 Heuristics

- SPEA was designed for and is suited to combinatorial and continuous function multiple objective optimization problem instances.
- A binary representation can be used for continuous function optimization problems in conjunction with classical genetic operators such as one-point crossover and point mutation.
- A k value of 1 may be used for efficiency whilst still providing useful results.
- The size of the archive is commonly smaller than the size of the population.
- There is a lot of room for implementation optimizations in density and Pareto dominance calculations.

7 Code Listing

Listing 1 provides an example of the Strength Pareto Evolutionary Algorithm 2 (SPEA2) implemented in the Ruby Programming Language. The demonstration problem is an instance of continuous multiple objective function optimization called SCH (problem one in [3]). The problem seeks the minimum of two functions: $f1 = \sum_{i=1}^{n} x_i^2$ and $f2 = \sum_{i=1}^{n}(x_i-2)^2$, $-10^3 \leq x_i \leq 10^3$ and $n = 1$. The optimal solution for this function are $x \in [0, 2]$. The algorithm is an implementation of SPEA2 based on the presentation by Zitzler, Laumanns, and Thiele [7]. The algorithm

Algorithm 1: Pseudo Code for the Strength Pareto Evolutionary Algorithm 2 (SPEA2).

Input: $Population_{size}$, $Archive_{size}$, ProblemSize, $P_{crossover}$, $P_{mutation}$
Output: Archive

Population $\leftarrow$ `InitializePopulation`($Population_{size}$, ProblemSize);
Archive $\leftarrow$ 0;
while *True* **do**
  for $S_i \in$ Population **do**
    $Si_{objectives} \leftarrow$ `CalculateObjectives`(S_i);
  end
  Union $\leftarrow$ Population + Archive;
  for $S_i \in$ Union **do**
    $Si_{raw} \leftarrow$ `CalculateRawFitness`(S_i, Union);
    $Si_{density} \leftarrow$ `CalculateSolutionDensity`(S_i, Union);
    $Si_{fitness} \leftarrow Si_{raw} + Si_{density}$;
  end
  Archive $\leftarrow$ `GetNonDominated`(Union);
  if `Size`(Archive) $< Archive_{size}$ **then**
    `PopulateWithRemainingBest`(Union, Archive, $Archive_{size}$);
  end
  else if `Size`(Archive) $> Archive_{size}$ **then**
    `RemoveMostSimilar`(Archive, $Archive_{size}$);
  end
  if `StopCondition`() **then**
    Archive $\leftarrow$ `GetNonDominated`(Archive);
    `Break`();
  else
    Selected $\leftarrow$ `SelectParents`(Archive, $Population_{size}$);
    Population $\leftarrow$ `CrossoverAndMutation`(Selected, $P_{crossover}$, $P_{mutation}$);
  end
end
return Archive;

uses a binary string representation (16 bits per objective function parameter) that is decoded using the binary coded decimal method and rescaled to the function domain. The implementation uses a uniform crossover operator and point mutations with a fixed mutation rate of $\frac{1}{L}$, where L is the number of bits in a solution's binary string.

```ruby
BITS_PER_PARAM = 16

def objective1(vector)
  return vector.inject(0.0) {|sum, x| sum + (x**2.0)}
end

def objective2(vector)
  return vector.inject(0.0) {|sum, x| sum + ((x-2.0)**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)].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 point_mutation(bitstring)
  child = ""
  bitstring.size.times do |i|
    bit = bitstring[i]
    child << ((rand()<1.0/bitstring.length.to_f) ? ((bit=='1') ? "0" : "1") : bit)
  end
  return child
end

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

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

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

def calculate_objectives(pop, search_space)
  pop.each do |p|
    p[:vector] = decode(p[:bitstring], search_space)
    p[:objectives] = []
    p[:objectives] << objective1(p[:vector])
    p[:objectives] << objective2(p[:vector])
  end
end

def dominates(p1, p2)
  p1[:objectives].each_with_index do |x,i|
    return false if x > p2[:objectives][i]
  end
  return true
end

def weighted_sum(x)
  return x[:objectives].inject(0.0) {|sum, x| sum+x}
end

def distance(c1, c2)
  sum = 0.0
```

```ruby
  c1.each_with_index {|x,i| sum += (c1[i]-c2[i])**2.0}
  return Math.sqrt(sum)
end

def calculate_dominated(pop)
  pop.each do |p1|
    p1[:dom_set] = pop.select {|p2| dominates(p1, p2) }
  end
end

def calculate_raw_fitness(p1, pop)
  return pop.inject(0.0) do |sum, p2|
    (dominates(p2, p1)) ? sum + p2[:dom_set].size.to_f : sum
  end
end

def calculate_density(p1, pop)
  pop.each {|p2| p2[:dist] = distance(p1[:objectives], p2[:objectives])}
  list = pop.sort{|x,y| x[:dist]<=>y[:dist]}
  k = Math.sqrt(pop.length).to_i
  return 1.0 / (list[k][:dist] + 2.0)
end

def calculate_fitness(pop, archive, search_space)
  calculate_objectives(pop, search_space)
  union = archive + pop
  calculate_dominated(union)
  union.each do |p1|
    p1[:raw_fitness] = calculate_raw_fitness(p1, union)
    p1[:density] = calculate_density(p1, union)
    p1[:fitness] = p1[:raw_fitness] + p1[:density]
  end
end

def environmental_selection(pop, archive, archive_size)
  union = archive + pop
  environment = union.select {|p| p[:fitness]<1.0}
  if environment.length < archive_size
    union.sort!{|x,y| x[:fitness]<=>y[:fitness]}
    union.each do |p|
      environment << p if p[:fitness] >= 1.0
      break if environment.length >= archive_size
    end
  elsif environment.length > archive_size
    begin
      k = Math.sqrt(environment.length).to_i
      environment.each do |p1|
        environment.each {|p2| p2[:dist] = distance(p1[:objectives], p2[:objectives])}
        list = environment.sort{|x,y| x[:dist]<=>y[:dist]}
        p1[:density] = list[k][:dist]
      end
      environment.sort!{|x,y| x[:density]<=>y[:density]}
      environment.shift
    end until environment.length >= archive_size
  end
  return environment
end

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

```

```ruby
def search(problem_size, search_space, max_gens, pop_size, archive_size, p_crossover)
  pop = Array.new(pop_size) do |i|
    {:bitstring=>random_bitstring(problem_size*BITS_PER_PARAM)}
  end
  gen, archive = 0, []
  begin
    calculate_fitness(pop, archive, search_space)
    archive = environmental_selection(pop, archive, archive_size)
    best = archive.sort{|x,y| weighted_sum(x)<=>weighted_sum(y)}.first
    puts ">gen=#{gen}, best: x=#{best[:vector]}, objs=#{best[:objectives].join(', ')}"
    if gen >= max_gens
      archive = archive.select {|p| p[:fitness]<1.0}
      break
    else
      selected = Array.new(pop_size){binary_tournament(archive)}
      pop = reproduce(selected, pop_size, p_crossover)
      gen += 1
    end
  end while true
  return archive
end

max_gens = 50
pop_size = 80
archive_size = 40
p_crossover = 0.90
problem_size = 1
search_space = Array.new(problem_size) {|i| [-1000, 1000]}

pop = search(problem_size, search_space, max_gens, pop_size, archive_size, p_crossover)
puts "done!"
```

Listing 1: Strength Pareto Evolutionary Algorithm 2 SPEA2) in the Ruby Programming Language

8 References

8.1 Primary Sources

Zitzler and Thiele introduced the Strength Pareto Evolutionary Algorithm as a technical report on a multiple objective optimization algorithm with elitism and clustering along the Pareto front [8]. The technical report was later published [9]. The Strength Pareto Evolutionary Algorithm was developed as a part of Zitzler PhD thesis [4]. Zitzler, Laumanns, and Thiele later extended SPEA to address some inefficiencies the approach, called SPEA2 that was released as a technical report [6] and later published [7]. SPEA2 provided a fine-grained fitness assignment, density estimation on the Pareto front, and an archive truncation operator.

8.2 Learn More

Zitzler, Laumanns, and Bleuler provide a tutorial on SPEA2 as a book chapter that considers the basics of multiple objective optimization, and the differences from SPEA and the other related Multiple Objective Evolutionary Algorithms [5].

9 Conclusions

This report described the Strength Pareto Evolutionary Algorithm as an evolutionary multiple objective optimization algorithm.

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 (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. 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.

[3] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[4] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Shaker Verlag, Aachen, Germany, 1999.

[5] E. Zitzler, M. Laumanns, and S. Bleuler. *Metaheuristics for Multiobjective Optimisation*, chapter A Tutorial on Evolutionary Multiobjective Optimization, pages 3–37. Springer, 2004.

[6] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland, May 2001.

[7] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. *Evolutionary Methods for Design Optimisation and Control*, pages 95–100, 2002.

[8] E. Zitzler and L. Thiele. An evolutionary algorithm for multiobjective optimization: The strength pareto approach. Technical Report 43, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland, May 1998.

[9] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.