Non-dominated Sorting Genetic Algorithm*

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

April 26, 2010
Technical Report: CA-TR-20100426-1

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 Non-dominated Sorting Genetic Algorithm using the standardized template.

Keywords: Clever, Algorithms, Description, Optimization, Non-dominated, Sorting, Genetic, 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 Non-dominated Sorting Genetic Algorithm using the standardized template.

2 Name

Non-dominated Sorting Genetic Algorithm, Nondominated Sorting Genetic Algorithm, Fast Elitist Non-dominated Sorting Genetic Algorithm, NSGA, NSGA-II, NSGAII

3 Taxonomy

The Non-dominated Sorting Genetic Algorithm is a Multiple Objective Optimization (MOO) algorithm and is an instance of an Evolutionary Algorithm (EA) from the field of Evolutionary Computation (EC). NSGA is an extension of the Genetic Algorithm (GA) for multiple objective function optimization. It is related to other Evolutionary Multiple Objective Optimization Algorithms (EMOO) (or Multiple Objective Evolutionary Algorithms MOEA) such as the Vector-Evaluated Genetic Algorithm (VEGA), Strength Pareto Evolutionary Algorithm (SPEA), and Pareto Archived Evolution Strategy (PAES). There are two versions of the algorithm, the classical NSGA and the updated and currently canonical form NSGA-II.

4 Strategy

The objective of the NSGA algorithm is to improve the adaptive fit of a population of candidate solutions to a Pareto front constrained by a set of objective functions. The algorithm uses an evolutionary process with surrogates for evolutionary operators including selection, genetic crossover, and genetic mutation. The population is sorted into a hierarchy of sub-populations based on the ordering of Pareto dominance. Similarity between members of each sub-group is evaluated on the Pareto front, and the resulting groups and similarity measures are used to promote a diverse front of non-dominated solutions.

5 Procedure

Algorithm 1 provides a pseudo-code listing of the Non-dominated Sorting Genetic Algorithm II (NSGA-II) for minimizing a cost function. The `SortByRankAndDistance` function orders the population into a hierarchy of non-dominated Pareto fronts. The `CrowdingDistanceAssignment` calculates the average distance between members of each front on the front itself. Refer to Deb et al. for a clear presentation of the pseudo code and explanation of these functions [6]. The `CrossoverAndMutation` function performs the classical crossover and mutation genetic operators of the Genetic Algorithm. Both the `SelectParentsByRankAndDistance` and `SortByRankAndDistance` functions discriminate members of the population first by rank (order of dominated precedence of the front to which the solution belongs) and then distance within the front (calculated by `CrowdingDistanceAssignment`).

6 Heuristics

- NSGA was designed for and is suited to continuous function multiple objective optimization problem instances.
- A binary representation can be used in conjunction with classical genetic operators such as one-point crossover and point mutation.
- A real-valued representation is recommended for continuous function optimization problems, in turn requiring representation specific genetic operators such as Simulated Binary Crossover (SBX) and polynomial mutation [3].

7 Code Listing

Listing 1 provides an example of the Non-dominated Sorting Genetic Algorithm II (NSGA-II) implemented in the Ruby Programming Language. The demonstration problem is an instance of continuous multiple objective function optimization called SCH (problem one in [6]). 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 NSGA-II based on the presentation by Deb, et al. [6]. The algorithm 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.

Algorithm 1: Pseudo Code for the Non-dominated Sorting Genetic Algorithm II.

```
Input: Population_size, ProblemSize, P_crossover, P_mutation
Output: S_best
Population ← InitializePopulation(Population_size, ProblemSize);
EvaluateAgainstObjectiveFunctions(Population);
FastNondominatedSort(Population);
Selected ← SelectParentsByRank(Population, Population_size);
Children ← CrossoverAndMutation(Selected, P_crossover, P_mutation);
while ¬StopCondition() do
    EvaluateAgainstObjectiveFunctions(Children);
    Union ← Merge(Population, Children);
    Fronts ← FastNondominatedSort(Union);
    Parents ← 0;
    Front_L ← 0;
    foreach Front_i ∈ Fronts do
        CrowdingDistanceAssignment(Front_i);
        if Size(Parents)+Size(Front_i) > Population_size then
            Front_L ← i;
            Break();
        else
            Parents ← Merge(Parents, Front_i);
        end
    end
    if Size(Parents)<Population_size then
        Front_L ← SortByRankAndDistance(Front_L);
        for P_1 to P_{Population_size−Size(LastFront)} do
            Parents ← Pi;
        end
    end
    Selected ← SelectParentsByRankAndDistance(Parents, Population_size);
    Population ← Children;
    Children ← CrossoverAndMutation(Selected, P_crossover, P_mutation);
end
return Children;
```

```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 fast_nondominated_sort(pop)
  fronts = Array.new(1){[]}
  pop.each do |p1|
    p1[:dom_count], p1[:dom_set] = 0, []
    pop.each do |p2|
      if dominates(p1, p2)
```

```ruby
      p1[:dom_set] << p2
    elsif dominates(p2, p1)
      p1[:dom_count] += 1
    end
  end
  if p1[:dom_count] == 0
    p1[:rank] = 0
    fronts.first << p1
  end
end
curr = 0
begin
  next_front = []
  fronts[curr].each do |p1|
    p1[:dom_set].each do |p2|
      p2[:dom_count] -= 1
      if p2[:dom_count] == 0
        p2[:rank] = (curr+1)
        next_front << p2
      end
    end
  end
  curr += 1
  fronts << next_front if !next_front.empty?
end while curr < fronts.length
return fronts
end

def calculate_crowding_distance(pop)
  pop.each {|p| p[:distance] = 0.0}
  num_obs = pop.first[:objectives].length
  num_obs.times do |i|
    pop.sort!{|x,y| x[:objectives][i]<=>y[:objectives][i]}
    min, max = pop.first[:objectives][i], pop.last[:objectives][i]
    range, inf = max-min, 1.0/0.0
    pop.first[:distance], pop.last[:distance] = inf, inf
    next if range == 0
    (1...(pop.length-2)).each do |j|
      pop[j][:distance] += (pop[j+1][:objectives][i] - pop[j-1][:objectives][i]) / range
    end
  end
end

def crowded_comparison_operator(x,y)
  return y[:distance]<=>x[:distance] if x[:rank] == y[:rank]
  return x[:rank]<=>y[:rank]
end

def better(x,y)
  if !x[:distance].nil? and x[:rank] == y[:rank]
    return (x[:distance]>y[:distance]) ? x : y
  end
  return (x[:rank]<y[:rank]) ? x : y
end

def select_parents(fronts, pop_size)
  fronts.each {|f| calculate_crowding_distance(f)}
  offspring = []
  last_front = 0
  fronts.each do |front|
    break if (offspring.length+front.length) > pop_size
    front.each {|p| offspring << p}
    last_front += 1
```

```ruby
  end
  if (remaining = pop_size-offspring.length) > 0
    fronts[last_front].sort! {|x,y| crowded_comparison_operator(x,y)}
    offspring += fronts[last_front][0...remaining]
  end
  return offspring
end

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

def search(problem_size, search_space, max_gens, pop_size, p_crossover)
  pop = Array.new(pop_size) do |i|
    {:bitstring=>random_bitstring(problem_size*BITS_PER_PARAM)}
  end
  calculate_objectives(pop, search_space)
  fast_nondominated_sort(pop)
  selected = Array.new(pop_size){better(pop[rand(pop_size)], pop[rand(pop_size)])}
  children = reproduce(selected, pop_size, p_crossover)
  calculate_objectives(children, search_space)
  max_gens.times do |gen|
    union = pop + children
    fronts = fast_nondominated_sort(union)
    offspring = select_parents(fronts, pop_size)
    selected = Array.new(pop_size){better(offspring[rand(pop_size)], offspring[rand(pop_size)])}
    pop = children
    children = reproduce(selected, pop_size, p_crossover)
    calculate_objectives(children, search_space)
    best = children.sort!{|x,y| weighted_sum(x)<=>weighted_sum(y)}.first
    best_s = "[x=#{best[:vector]}, objs=#{best[:objectives].join(', ')}]"
    puts " > gen=#{gen+1}, fronts=#{fronts.length}, best=#{best_s}"
  end
  return children
end

max_gens = 50
pop_size = 100
p_crossover = 0.98
problem_size = 1
search_space = Array.new(problem_size) {|i| [-1000, 1000]}

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

Listing 1: Non-dominated Sorting Genetic Algorithm II (NSGA-II) in the Ruby Programming Language

8 References

8.1 Primary Sources

Srinivas and Deb proposed the NSGA algorithm inspired by Goldberg's notion of a non-dominated sorting procedure [9]. Goldberg proposed a non-dominated sorting procedure in his book in considering the biases in the Pareto optimal solutions provided by VEGA [7]. Srinivas and Deb's NSGA used the sorting procedure as a ranking selection method, and a fitness sharing niching method to maintain stable sub-populations across the Pareto front. Deb, et al. later extended NSGA to address three criticism of the approach: i) the $O(mN^3)$ time complexity, the lack of elitism, and the need for a sharing parameter for the fitness sharing niching method [5, 6].

8.2 Learn More

Deb provides in depth coverage of Evolutionary Multiple Objective Optimization algorithms in his book, including a detailed description of the NSGA in Chapter 5 [4].

9 Conclusions

This report described the Non-dominated Sorting Genetic Algorithm as a Multiple Objective Evolutionary Algorithm (MOEA). The research for this report resulted in the identification of a large number of additional MOEAs concisely listed in Konak, Coit, and Smith's paper [8] (please refer for references). They are as follows: Multi-objective Genetic Algorithm (MOGA), Niched Pareto Genetic Algorithm (NPGA), Weight-based Genetic Algorithm (WBGA), Random Weighted Genetic Algorithm (RWGA), Pareto-Archived Evolution Strategy (PAES), Pareto Envelope-based Selection Algorithm (PESA), Region-based Selection in Evolutionary Multiobjective Optimization (PESA-II), Multi-objective Evolutionary Algorithm (MEA), Micro-GA, Rank-Density Based Genetic Algorithm (RDGA), and the Dynamic Multi-objective Evolutionary Algorithm (DMOEA).

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] K. Deb and R. B. Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9:115–148, 1995.

[4] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley and Sons, 2001.

[5] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. *Parallel Problem Solving from Nature PPSN VI*, 1917:849–858, 2000.

[6] 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.

[7] David Edward Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, 1989.

[8] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*, 91:992–1007, 2006.

[9] N. Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.