Scatter Search*

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

February 16, 2010
Technical Report: CA-TR-20100216-1

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 Scatter Search algorithm using the standardized template.

Keywords: `Clever, Algorithms, Description, Optimization, Scatter, Search`

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 Scatter Search algorithm using the standardized template.

2 Name

Scatter Search, SS

3 Taxonomy

Scatter search is a Metaheuristic and a Global Optimization algorithm. It is also sometimes associated with the field of Evolutionary Computation given the use of a population and recombination in the structure of the technique. Scatter Search is a sibling of Tabu Search, developed by the same author and based on similar origins.

4 Strategy

The objective of Scatter Search is to maintain a set of diverse and high-quality candidate solutions. The principle of the approach is that useful information about the global optima is stored in a diverse and elite set of solutions (the reference set) and that recombining samples from the set can exploit this information. The strategy involves an iterative process, where a population of diverse and high-quality candidate solutions that are partitioned into subsets and linearly recombined to create weighted centroids of sample-based neighborhoods. The results of recombination are refined using an embedded heuristic and assessed in the context of the reference set as to whether or not they are retained.

5 Procedure

Algorithm 1 provides a pseudo-code listing of the Scatter Search algorithm for minimizing a cost function. The procedure is based on the abstract form presented by Glover as a template for the general class of technique [5], with influences from an application of the technique to function optimization by Glover [5].

Algorithm 1: Pseudo Code for the Scatter Search algorithm.

```
Input: DiverseSet_size, ReferenceSet_size
Output: ReferenceSet
InitialSet ← ConstructInitialSolution(DiverseSet_size);
RefinedSet ← 0;
for S_i ∈ InitialSet do
    RefinedSet ← LocalSearch(S_i);
end
ReferenceSet ← SelectInitialReferenceSet(ReferenceSet_size);
while ¬ StopCondition() do
    Subsets ← SelectSubset(ReferenceSet);
    CandidateSet ← 0;
    for Subset_i ∈ Subsets do
        RecombinedCandidates ← RecombineMembers(Subset_i);
        for S_i ∈ RecombinedCandidates do
            CandidateSet ← LocalSearch(S_i);
        end
    end
    ReferenceSet ← Select(ReferenceSet, CandidateSet, ReferenceSet_size);
end
return ReferenceSet;
```

6 Heuristics

- Scatter search is suitable for both discrete domains such as combinatorial optimization as well as continuous domains such as non-linear programming (continuous function optimization).
- Small set sizes are preferred for the `ReferenceSet`, such as 10 or 20 members.
- Subset sizes can be 2,3,4 or more members that are all recombined to produce viable candidate solutions within the neighborhood of the members of the subset.

- Each subset should comprise at least one member added to the set in the previous algorithm iteration.
- The Local Search procedure should be a problem-specific improvement heuristic.
- The selection of members for the `ReferenceSet` at the end of each iteration favors solutions with higher quality and may also promote diversity.
- The `ReferenceSet` may be updated at the end of an iteration, or dynamically as candidates are created (a so-called steady-state population in some evolutionary computation literature).
- A lack of changes to the `ReferenceSet` may be used as a signal to stop the current search, and potentially restart the search with a newly initialized `ReferenceSet`.

7 Code Listing

Listing 1 provides an example of the Scatter Search algorithm implemented in the Ruby Programming Language. The example 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 $(v_1, \ldots, v_n) = 0.0$.

The algorithm is an implementation of Scatter Search as described in an application of the technique to unconstrained non-linear optimization by Glover [8]. The seeds for initial solutions are generated as random vectors, as opposed to stratified samples. The example was further simplified by not including a restart strategy, and the exclusion of diversity maintenance in the `ReferenceSet`. A stochastic local search algorithm is used as the embedded heuristic that uses a stochastic step size in the range of half a percent of the search space.

```
NUM_ITERATIONS = 100
PROBLEM_SIZE = 3
SEARCH_SPACE = Array.new(PROBLEM_SIZE) {|i| [-5, +5]}
STEP_SIZE = (SEARCH_SPACE[0][1]-SEARCH_SPACE[0][0])*0.005
LS_MAX_NO_IMPROVEMENTS = 30
REF_SET_SIZE = 10
DIVERSE_SET_SIZE = 20
NO_ELITE = 5

def cost(candidate_vector)
  return candidate_vector.inject(0) {|sum, x| sum + (x ** 2.0)}
end

def random_solution(problemSize, searchSpace)
  return Array.new(problemSize) do |i|
    searchSpace[i][0] + ((searchSpace[i][1] - searchSpace[i][0]) * rand)
  end
end

def take_step(currentPosition, searchSpace, stepSize)
  step = []
  currentPosition.length.times do |i|
    max, min = currentPosition[i]+stepSize, currentPosition[i]-stepSize
    max = searchSpace[i][1] if max > searchSpace[i][1]
    min = searchSpace[i][0] if min < searchSpace[i][0]
    step << min + ((max - min) * rand)
  end
  return step
end

def local_search(best, searchSpace, maxNoImprovements, stepSize)
```

```ruby
  noImprovements = 0
  begin
    candidate = {}
    candidate[:vector] = take_step(best[:vector], searchSpace, stepSize)
    candidate[:cost] = cost(candidate[:vector])
    if candidate[:cost] < best[:cost]
      noImprovements, best = 0, candidate
    else
      noImprovements += 1
    end
  end until noImprovements >= maxNoImprovements
  return best
end

def construct_initial_set(problemSize, searchSpace, divSetSize, maxNoImprovements, stepSize)
  diverseSet = []
  begin
    candidate = {}
    candidate[:vector] = random_solution(problemSize, searchSpace)
    candidate[:cost] = cost(candidate[:vector])
    candidate = local_search(candidate, searchSpace, maxNoImprovements, stepSize)
    diverseSet << candidate if !diverseSet.any? {|x| x[:vector]==candidate[:vector]}
  end until diverseSet.length == divSetSize
  return diverseSet
end

def euclidean(v1, v2)
  sum = 0.0
  v1.each_with_index {|v, i| sum += (v**2.0 - v2[i]**2.0) }
  sum = sum + (0.0-sum) if sum < 0.0
  return Math.sqrt(sum)
end

def distance(vector1, referenceSet)
  sum = 0.0
  referenceSet.each do |s|
    sum += euclidean(vector1, s[:vector])
  end
  return sum
end

def diversify(diverseSet, numElite, refSetSize)
  diverseSet.sort!{|x,y| x[:cost] <=> y[:cost]}
  referenceSet = Array.new(numElite){|i| diverseSet[i]}
  remainder = diverseSet - referenceSet
  remainder.sort!{|x,y| distance(y[:vector], referenceSet) <=> distance(x[:vector],
      referenceSet)}
  referenceSet = referenceSet + remainder[0..(refSetSize-referenceSet.length)]
  return referenceSet, referenceSet[0]
end

def select_subsets(referenceSet)
  additions = referenceSet.select{|c| c[:new]}
  remainder = referenceSet - additions
  remainder = additions if remainder.empty?
  subsets = []
  additions.each{|a| remainder.each{|r| subsets << [a,r] if a!=r}}
  return subsets
end

def recombine(subset, problemSize, searchSpace)
  a, b = subset
  d = rand(euclidean(a[:vector], b[:vector]))/2.0
```

```ruby
  children = []
  subset.each do |p|
    step = (rand<0.5) ? +d : -d
    child = {}
    child[:vector] = Array.new(problemSize){|i| p[:vector][i]+step}
    child[:vector].each_with_index {|m,i| child[:vector][i]=searchSpace[i][0] if
        m<searchSpace[i][0]}
    child[:vector].each_with_index {|m,i| child[:vector][i]=searchSpace[i][1] if
        m>searchSpace[i][1]}
    child[:cost] = cost(child[:vector])
    children << child
  end
  return children
end

def search(problemSize, searchSpace, numIterations, refSetSize, divSetSize, maxNoImprovements,
    stepSize, noElite)
  diverseSet = construct_initial_set(problemSize, searchSpace, divSetSize, maxNoImprovements,
      stepSize)
  referenceSet, best = diversify(diverseSet, noElite, refSetSize)
  referenceSet.each{|c| c[:new] = true}
  numIterations.times do |iter|
    wasChange = false
    subsets = select_subsets(referenceSet)
    referenceSet.each{|c| c[:new] = false}
    subsets.each do |subset|
      candidates = recombine(subset, problemSize, searchSpace)
      improved = Array.new(candidates.length) {|i| local_search(candidates[i], searchSpace,
          maxNoImprovements, stepSize)}
      improved.each do |c|
        if !referenceSet.any? {|x| x[:vector]==c[:vector]}
          c[:new] = true
          referenceSet.sort!{|x,y| x[:cost] <=> y[:cost]}
          if c[:cost]<referenceSet.last[:cost]
            referenceSet.delete(referenceSet.last)
            referenceSet << c
            wasChange = true
          end
        end
      end
    end
    referenceSet.sort!{|x,y| x[:cost] <=> y[:cost]}
    best = referenceSet[0] if referenceSet[0][:cost] < best[:cost]
    puts " > iteration #{(iter+1)}, best: c=#{best[:cost]}"
    break if !wasChange
  end
  return best
end

best = search(PROBLEM_SIZE, SEARCH_SPACE, NUM_ITERATIONS, REF_SET_SIZE, DIVERSE_SET_SIZE,
    LS_MAX_NO_IMPROVEMENTS, STEP_SIZE, NO_ELITE)
puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
```

Listing 1: Scatter Search algorithm in the Ruby Programming Language

8 References

8.1 Primary Sources

A form of the Scatter Search algorithm was proposed by Glover for integer programming [3], based on Glover's earlier work on surrogate constraints. The approach remained idle until it was revisited by Glover and combined with Tabu Search [4]. The modern canonical reference of the approach was proposed by Glover who provides a abstract template of the procedure that may be specialized for a given application domain [5].

8.2 Learn More

The primary reference for the approach is the book by Laguna and Martí that reviews the principles of the approach in detail and presents tutorials on applications of the approach on standard problems using the C programming language [9]. There are many review articles and chapters on Scatter Search that may be used to supplement an understanding of the approach, such as a detailed review chapter by Glover [6], a review of the fundamentals of the approach and its relationship to an abstraction called 'path linking' by Glover, Laguna, and Martí [7], and a modern overview of the technique by Martí, Lagunab, and Glover [10].

9 Conclusions

This report described the Scatter Search algorithm as a procedure that maintains a diverse and high-quality set of candidate solutions that are recombined to create new candidate solutions.

This preparation of this report was particularly difficult compared to recent reports. The reason for this (as proposed by the author) was because of the vagueness of the description of the algorithm in the literature and the over use of poor pseudo code listings in review articles and chapters. It is hoped that experts in the field may be consulted and/or improved descriptions of the approach can be identified.

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] 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] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977.

[4] F. Glover. Tabu search for nonlinear and parametric optimization (with links to genetic algorithms). *Discrete Applied Mathematics*, 49:231–255, 1994.

[5] F. Glover. *Artificial Evolution*, chapter A Template For Scatter Search And Path Relinking, page 13. Sprinter, 1998.

[6] F. Glover. *New Ideas in Optimization*, chapter Scatter search and path relinking, pages 297–316. McGraw-Hill Ltd., 1999.

[7] F. Glover, M. Laguna, and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39(3):653–684, 2000.

[8] F. Glover, M. Laguna, and R. Martí. *Advances in Evolutionary Computation: Theory and Applications*, chapter Scatter Search, pages 519–537. Springer-Verlag, 2003.

[9] Manuel Laguna and Rafael Martí. *Scatter search: methodology and implementations in C*. Kluwer Academic Publishers, 2003.

[10] Rafael Martí, Manuel Lagunab, and Fred Glover. Principles of scatter search. *European Journal of Operational Research*, 169(1):359–372, 2006.