# Chapter 6

# Swarm Algorithms

## 6.1 Overview

This chapter describes Swarm Algorithms.

### 6.1.1 Swarm Intelligence

Swarm intelligence is the study of computational systems inspired by the 'collective intelligence'. Collective Intelligence emerges through the cooperation of large numbers of homogeneous agents in the environment. Examples include schools of fish, flocks of birds, and colonies of ants. Such intelligence is decentralized, self-organizing and distributed through out an environment. In nature such systems are commonly used to solve problems such as effective foraging for food, prey evading, or colony re-location. The information is typically stored throughout the participating homogeneous agents, or is stored or communicated in the environment itself such as through the use of pheromones in ants, dancing in bees, and proximity in fish and birds.

The paradigm consists of two dominant sub-fields 1) *Ant Colony Optimization* that investigates probabilistic algorithms inspired by the stigmergy and foraging behavior of ants, and 2) *Particle Swarm Optimization* that investigates probabilistic algorithms inspired by the flocking, schooling and herding. Like evolutionary computation, swarm intelligence 'algorithms' or 'strategies' are considered adaptive strategies and are typically applied to search and optimization domains.

### 6.1.2 References

Seminal books on the field of Swarm Intelligence include "*Swarm Intelligence*" by Kennedy, Eberhart and Shi [10], and "*Swarm Intelligence: From Natural to Artificial Systems*" by Bonabeau, Dorigo, and Theraulaz [3]. Another excellent text book on the area is "*Fundamentals of Computational Swarm*

*Intelligence*" by Engelbrecht [7]. The seminal book reference for the field of Ant Colony Optimization is "*Ant Colony Optimization*" by Dorigo and Stützle [6].

### 6.1.3 Extensions

There are many other algorithms and classes of algorithm that were not described from the field of Swarm Intelligence, not limited to:

- **Ant Algorithms**: such as Max-Min Ant Systems [15] Rank-Based Ant Systems [4], Elitist Ant Systems [5], Hyper Cube Ant Colony Optimization [2] Approximate Nondeterministic Tree-Search (ANTS) [12] and Multiple Ant Colony System [8].
- **Bee Algorithms**: such as Bee System and Bee Colony Optimization [11], the Honey Bee Algorithm [16], and Artificial Bee Colony Optimization [1, 9].
- **Other Social Insects**: algorithms inspired by other social insects besides ants and bees, such as the Firey Algorithm [18] and the Wasp Swarm Algorithm [14].
- **Extensions to Particle Swarm**: such as Repulsive Particle Swarm Optimization [17].
- **Bacteria Algorithms**: such as the Bacteria Chemotaxis Algorithm [13].

### 6.1.4 Bibliography

[1] B. Basturk and D. Karaboga. An artificial bee colony (ABC) algorithm for numeric function optimization. In *IEEE Swarm Intelligence Symposium*, 2006.

[2] C. Blum, A. Roli, and M. Dorigo. HC–ACO: The hyper-cube framework for ant colony optimization. In *Proceedings of the Fourth Metaheuristics International Conference*, volume 1, pages 399–403, 2001.

[3] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press US, 1999.

[4] B. Bullnheimer, R. F. Hartl, and C. Strauss. A new rank based version of the ant system: A computational study. *Central European Journal for Operations Research and Economics*, 7(1):25–38, 1999.

[5] M. Dorigo. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and CyberneticsPart B*, 1:1–13, 1996.

[6] M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, 2004.

[7] A. P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence.* Wiley & Sons, 2006.

[8] L. M. Gambardella, E. Taillard, and G. Agazzi. *New Ideas in Optimization*, chapter MACS–VRPTW: A Multiple Ant Colony System for Vehicle Routing Problems with Time Windows, pages 63–76. McGraw-Hill, 1999.

[9] D. Karaboga. An idea based on honey bee swarm for numerical optimization. Technical Report TR06, Erciyes University, Engineering Faculty, Computer Engineering Department, 2005.

[10] J. Kennedy, R. C. Eberhart, and Y. Shi. *Swarm Intelligence.* Morgan Kaufmann, 2001.

[11] P. Lučić and D. Teodorović. Bee system: modeling combinatorial optimization transportation engineering problems by swarm intelligence. In *Preprints of the TRISTAN IV Triennial Symposium on Transportation Analysis*, pages 441–445, 2001.

[12] V. Maniezzo. Approximate nondeterministic tree-search procedures for the quadratic assignment problem. *INFORMS Journal on Computing*, 11(4):358–369, 1999.

[13] S. D. Müller, J. Marchetto, S. Airaghi, and P. Koumoutsakos. Optimization based on bacterial chemotaxis. *IEEE Transactions on Evolutionary Computation*, 6(1):16–29, 2002.

[14] P. C. Pinto, T. A. Runkler, and J. M. Sousa. Wasp swarm algorithm for dynamic max-sat problems. In *Proceedings of the 8th international conference on Adaptive and Natural Computing Algorithms, Part I*, pages 350–357. Springer, 2007.

[15] T. Stützle and H. H. Hoos. MAX–MIN ant system, future generation computer systems. *Future Generation Computer Systems*, 16:889–914, 2000.

[16] C. Tovey. The honey bee algorithm: A biological inspired approach to internet server optimization. *Engineering Enterprise, the Alumni Magazine for ISyE at Georgia Institute of Technology*, pages 3–15, 2004.

[17] O Urfalioglu. Robust estimation of camera rotation, translation and focal length at high outlier rates. In *Proceedings of the 1st Canadian Conference on Computer and Robot Vision*, 2004.

[18] X. S. Yang. *Nature-Inspired Metaheuristic Algorithms.* Luniver Press, 2008.

## 6.2 Particle Swarm Optimization

*Particle Swarm Optimization, PSO.*

### 6.2.1 Taxonomy

Particle Swarm Optimization belongs to the field of Swarm Intelligence and Collective Intelligence and is a sub-field of Computational Intelligence. Particle Swarm Optimization is related to other Swarm Intelligence algorithms such as Ant Colony Optimization and it is a baseline algorithm for many variations, too numerous to list.

### 6.2.2 Inspiration

Particle Swarm Optimization is inspired by the social foraging behavior of some animals such as flocking behavior of birds and the schooling behavior of fish.

### 6.2.3 Metaphor

Particles in the swarm fly through an environment following the fitter members of the swarm and generally biasing their movement toward historically good areas of their environment.

### 6.2.4 Strategy

The goal of the algorithm is to have all the particles locate the optima in a multi-dimensional hyper-volume. This is achieved by assigning initially random positions to all particles in the space and small initial random velocities. The algorithm is executed like a simulation, advancing the position of each particle in turn based on its velocity, the best known global position in the problem space and the best position known to a particle. The objective function is sampled after each position update. Over time, through a combination of exploration and exploitation of known good positions in the search space, the particles cluster or converge together around an optima, or several optima.

### 6.2.5 Procedure

The Particle Swarm Optimization algorithm is comprised of a collection of particles that move around the search space influenced by their own best past location and the best past location of the whole swarm or a close neighbor. Each iteration a particle's velocity is updated using:

$$v_i(t+1) = v_i(t) + \big(c_1 \times rand() \times (p_i^{best} - p_i(t))\big) + \big(c_2 \times rand() \times (p_{gbest} - p_i(t))\big)$$

where $v_i(t+1)$ is the new velocity for the $i^{th}$ particle, $c_1$ and $c_2$ are the weighting coefficients for the personal best and global best positions respectively, $p_i(t)$ is the $i^{th}$ particle's position at time $t$, $p_i^{best}$ is the $i^{th}$ particle's best known position, and $p_{gbest}$ is the best position known to the swarm. The $rand()$ function generate a uniformly random variable $\in [0,1]$. Variants on this update equation consider best positions within a particles local neighborhood at time $t$.

A particle's position is updated using:

$$p_i(t+1) = p_i(t) + v_i(t) \tag{6.1}$$

Algorithm 6.2.1 provides a pseudocode listing of the Particle Swarm Optimization algorithm for minimizing a cost function.

### 6.2.6 Heuristics

- The number of particles should be low, around 20-40
- The speed a particle can move (maximum change in its position per iteration) should be bounded, such as to a percentage of the size of the domain.
- The learning factors (biases towards global and personal best positions) should be between 0 and 4, typically 2.
- A local bias (local neighborhood) factor can be introduced where neighbors are determined based on Euclidean distance between particle positions.
- Particles may leave the boundary of the problem space and may be penalized, be reflected back into the domain or biased to return back toward a position in the problem domain. Alternatively, a wrapping strategy may be used at the edge of the domain creating a loop, torrid or related geometrical structures at the chosen dimensionality.
- An inertia or momentum coefficient can be introduced to limit the change in velocity.

### 6.2.7 Code Listing

Listing 6.1 provides an example of the Particle Swarm Optimization algorithm implemented in the Ruby Programming Language. The demonstration

**Algorithm 6.2.1**: Pseudocode for PSO.

```
Input: ProblemSize, $Population_{size}$
Output: $P_{g\_best}$
1  Population ← ∅;
2  $P_{g\_best} \leftarrow \emptyset$;
3  for $i = 1$ to $Population_{size}$ do
4      $P_{velocity}$ ← RandomVelocity();
5      $P_{position}$ ← RandomPosition($Population_{size}$);
6      $P_{cost}$ ← Cost($P_{position}$);
7      $P_{p\_best} \leftarrow P_{position}$;
8      if $P_{cost} \leq P_{g\_best}$ then
9          $P_{g\_best} \leftarrow P_{p\_best}$;
10     end
11 end
12 while ¬StopCondition() do
13     foreach $P \in$ Population do
14         $P_{velocity}$ ← UpdateVelocity($P_{velocity}$, $P_{g\_best}$, $P_{p\_best}$);
15         $P_{position}$ ← UpdatePosition($P_{position}$, $P_{velocity}$);
16         $P_{cost}$ ← Cost($P_{position}$);
17         if $P_{cost} \leq P_{p\_best}$ then
18             $P_{p\_best} \leftarrow P_{position}$;
19             if $P_{cost} \leq P_{g\_best}$ then
20                 $P_{g\_best} \leftarrow P_{p\_best}$;
21             end
22         end
23     end
24 end
25 return $P_{g\_best}$;
```

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 a conservative version of Particle Swarm Optimization based on the seminal papers. The implementation limits the velocity at a pre-defined maximum, and bounds particles to the search space, reflecting their movement and velocity if the bounds of the space are exceeded. Particles are influenced by the best position found as well as their own personal best position. Natural extensions may consider limiting velocity with an inertia coefficient and including a neighborhood function for the particles.

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

```

```ruby
def random_vector(minmax)
  return Array.new(minmax.size) do |i|
    minmax[i][0] + ((minmax[i][1] - minmax[i][0]) * rand())
  end
end

def create_particle(search_space, vel_space)
  particle = {}
  particle[:position] = random_vector(search_space)
  particle[:cost] = objective_function(particle[:position])
  particle[:b_position] = Array.new(particle[:position])
  particle[:b_cost] = particle[:cost]
  particle[:velocity] = random_vector(vel_space)
  return particle
end

def get_global_best(population, current_best=nil)
  population.sort{|x,y| x[:cost] <=> y[:cost]}
  best = population.first
  if current_best.nil? or best[:cost] <= current_best[:cost]
    current_best = {}
    current_best[:position] = Array.new(best[:position])
    current_best[:cost] = best[:cost]
  end
  return current_best
end

def update_velocity(particle, gbest, max_v, c1, c2)
  particle[:velocity].each_with_index do |v,i|
    v1 = c1 * rand() * (particle[:b_position][i] - particle[:position][i])
    v2 = c2 * rand() * (gbest[:position][i] - particle[:position][i])
    particle[:velocity][i] = v + v1 + v2
    particle[:velocity][i] = max_v if particle[:velocity][i] > max_v
    particle[:velocity][i] = -max_v if particle[:velocity][i] < -max_v
  end
end

def update_position(part, bounds)
  part[:position].each_with_index do |v,i|
    part[:position][i] = v + part[:velocity][i]
    if part[:position][i] > bounds[i][1]
      part[:position][i]=bounds[i][1]-(part[:position][i]-bounds[i][1]).abs
      part[:velocity][i] *= -1.0
    elsif part[:position][i] < bounds[i][0]
      part[:position][i]=bounds[i][0]+(part[:position][i]-bounds[i][0]).abs
      part[:velocity][i] *= -1.0
    end
  end
end

def update_best_position(particle)
  return if particle[:cost] > particle[:b_cost]
  particle[:b_cost] = particle[:cost]
  particle[:b_position] = Array.new(particle[:position])
end

```

```ruby
def search(max_gens, search_space, vel_space, pop_size, max_vel, c1, c2)
  pop = Array.new(pop_size) {create_particle(search_space, vel_space)}
  gbest = get_global_best(pop)
  max_gens.times do |gen|
    pop.each do |particle|
      update_velocity(particle, gbest, max_vel, c1, c2)
      update_position(particle, search_space)
      particle[:cost] = objective_function(particle[:position])
      update_best_position(particle)
    end
    gbest = get_global_best(pop, gbest)
    puts " > gen #{gen+1}, fitness=#{gbest[:cost]}"
  end
  return gbest
end

if __FILE__ == $0
  # problem configuration
  problem_size = 2
  search_space = Array.new(problem_size) {|i| [-5, 5]}
  # algorithm configuration
  vel_space = Array.new(problem_size) {|i| [-1, 1]}
  max_gens = 100
  pop_size = 50
  max_vel = 100.0
  c1, c2 = 2.0, 2.0
  # execute the algorithm
  best = search(max_gens, search_space, vel_space, pop_size, max_vel, c1,c2)
  puts "done! Solution: f=#{best[:cost]}, s=#{best[:position].inspect}"
end
```

Listing 6.1: Particle Swarm Optimization in Ruby

## 6.2.8 References

### Primary Sources

Particle Swarm Optimization was described as a stochastic global optimization method for continuous functions in 1995 by Eberhart and Kennedy [1, 3]. This work was motivated as an optimization method loosely based on the flocking behavioral models of Reynolds [7]. Early works included the introduction of inertia [8] and early study of social topologies in the swarm by Kennedy [2].

### Learn More

Poli, Kennedy, and Blackwell provide a modern overview of the field of PSO with detailed coverage of extensions to the baseline technique [6]. Poli provides a meta-analysis of PSO publications that focus on the application the technique, providing a systematic breakdown on application areas [5]. An excellent book on Swarm Intelligence in general with detailed coverage of

Particle Swarm Optimization is "Swarm Intelligence" by Kennedy, Eberhart, and Shi [4].

### 6.2.9 Bibliography

[1] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the sixth international symposium on micro machine and human science*, pages 39–43, 1995.

[2] J. Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In *Proceedings of the 1999 Congress on Evolutionary Computation*, 1999.

[3] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proceedings IEEE int'l conf. on neural networks Vol. IV*, pages 1942–1948, 1995.

[4] J. Kennedy, R. C. Eberhart, and Y. Shi. *Swarm Intelligence.* Morgan Kaufmann, 2001.

[5] R. Poli. Analysis of the publications on the applications of particle swarm optimisation. *Journal of Artificial Evolution and Applications*, 1:1–10, 2008.

[6] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization an overview. *Swarm Intelligence*, 1:33–57, 2007.

[7] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, 1987.

[8] Y. Shi and R. C. Eberhart. A modified particle swarm optimizers. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 69–73, 1998.

## 6.3 Ant System

*Ant System, AS, Ant Cycle.*

### 6.3.1 Taxonomy

The Ant System algorithm is an example of an Ant Colony Optimization method from the field of Swarm Intelligence, Metaheuristics and Computational Intelligence. Ant System was originally the term used to refer to a range of Ant based algorithms, where the specific algorithm implementation was referred to as Ant Cycle. The so-called Ant Cycle algorithm is now canonically referred to as Ant System. The Ant System algorithm is the baseline Ant Colony Optimization method for popular extensions such as Elite Ant System, Rank-based Ant System, Max-Min Ant System, and Ant Colony System.

### 6.3.2 Inspiration

The Ant system algorithm is inspired by the foraging behavior of ants, specifically the pheromone communication between ants regarding a good path between the colony and a food source in an environment. This mechanism is called stigmergy.

### 6.3.3 Metaphor

Ants initially wander randomly around their environment. Once food is located an ant will begin laying down pheromone in the environment. Numerous trips between the food and the colony are performed and if the same route is followed that leads to food then additional pheromone is laid down. Pheromone decays in the environment, so that older paths are less likely to be followed. Other ants may discover the same path to the food and in turn may follow it and also lay down pheromone. A positive feedback process routes more and more ants to productive paths that are in turn further refined through use.

### 6.3.4 Strategy

The objective of the strategy is to exploit historic and heuristic information to construct candidate solutions and fold the information learned from constructing solutions into the history. Solutions are constructed one discrete piece at a time in a probabilistic step-wise manner. The probability of selecting a component is determined by the heuristic contribution of the component to the overall cost of the solution and the quality of solutions from which the component has historically known to have been included. History is updated proportional to the quality of candidate solutions and is uniformly decreased ensuring the most recent and useful information is retained.

## 6.3.5 Procedure

Algorithm 6.3.1 provides a pseudocode listing of the main Ant System algorithm for minimizing a cost function. The pheromone update process is described by a single equation that combines the contributions of all candidate solutions with a decay coefficient to determine the new pheromone value, as follows:

$$\tau_{i,j} \leftarrow (1 - \rho) \times \tau_{i,j} + \sum_{k=1}^{m} \Delta_{i,j}^{k} \tag{6.2}$$

where $\tau_{i,j}$ represents the pheromone for the component (graph edge) $(i, j)$, $\rho$ is the decay factor, $m$ is the number of ants, and $\sum_{k=1}^{m} \Delta_{i,j}^{k}$ is the sum of $\frac{1}{S_{cost}}$ (maximizing solution cost) for those solutions that include component $i, j$. The Pseudocode listing shows this equation as an equivalent as a two step process of decay followed by update for simplicity.

The probabilistic step-wise construction of solution makes use of both history (pheromone) and problem-specific heuristic information to incrementally construction a solution piece-by-piece. Each component can only be selected if it has not already been chosen (for most combinatorial problems), and for those components that can be selected from (given the current component $i$), their probability for selection is defined as:

$$P_{i,j} \leftarrow \frac{\tau_{i,j}^{\alpha} \times \eta_{i,j}^{\beta}}{\sum_{k=1}^{c} \tau_{i,k}^{\alpha} \times \eta_{i,k}^{\beta}} \tag{6.3}$$

where $\eta_{i,j}$ is the maximizing contribution to the overall score of selecting the component (such as $\frac{1.0}{distance_{i,j}}$ for the Traveling Salesman Problem), $\alpha$ is the heuristic coefficient, $\tau_{i,j}$ is the pheromone value for the component, $\beta$ is the history coefficient, and $c$ is the set of usable components.

## 6.3.6 Heuristics

- The Ant Systems algorithm was designed for use with combinatorial problems such as the TSP, knapsack problem, quadratic assignment problems, graph coloring problems and many others.

- The history coefficient ($\alpha$) controls the amount of contribution history plays in a components probability of selection and is commonly set to 1.0.

**Algorithm 6.3.1**: Pseudocode for Ant System.

```
Input: ProblemSize, $Population_{size}$, $m$, $\rho$, $\alpha$, $\beta$
Output: $P_{best}$
1  $P_{best}$ ← CreateHeuristicSolution(ProblemSize);
2  $Pbest_{cost}$ ← Cost($S_h$);
3  Pheromone ← InitializePheromone($Pbest_{cost}$);
4  while ¬StopCondition() do
5      Candidates ← ∅;
6      for $i = 1$ to $m$ do
7          $S_i$ ← ProbabilisticStepwiseConstruction(Pheromone,
           ProblemSize, $\alpha$, $\beta$);
8          $Si_{cost}$ ← Cost($S_i$);
9          if $Si_{cost} \leq Pbest_{cost}$ then
10             $Pbest_{cost}$ ← $Si_{cost}$;
11             $P_{best}$ ← $S_i$;
12         end
13         Candidates ← $S_i$;
14     end
15     DecayPheromone(Pheromone, $\rho$);
16     foreach $S_i$ ∈ Candidates do
17         UpdatePheromone(Pheromone, $S_i$, $Si_{cost}$);
18     end
19 end
20 return $P_{best}$;
```

- The heuristic coefficient ($\beta$) controls the amount of contribution problem-specific heuristic information plays in a components probability of selection and is commonly between 2 and 5, such as 2.5.
- The decay factor ($\rho$) controls the rate at which historic information is lost and is commonly set to 0.5.
- The total number of ants ($m$) is commonly set to the number of components in the problem, such as the number of cities in the TSP.

### 6.3.7 Code Listing

Listing 6.2 provides an example of the Ant System algorithm implemented in the Ruby Programming Language. The algorithm is applied to the Berlin52 instance of the Traveling Salesman Problem (TSP), taken from the TSPLIB. The problem seeks a permutation of the order to visit cities (called a tour) that minimized the total distance traveled. The optimal tour distance for Berlin52 instance is 7542 units. Some extensions to the algorithm implementation for speed improvements may consider pre-calculating a distance matrix for all the cities in the problem, and pre-computing a probability matrix for choices during the probabilistic step-wise construction of tours.

```ruby
def euc_2d(c1, c2)
  Math.sqrt((c1[0] - c2[0])**2.0 + (c1[1] - c2[1])**2.0).round
end

def cost(permutation, cities)
  distance =0
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.size-1) ? permutation[0] : permutation[i+1]
    distance += euc_2d(cities[c1], cities[c2])
  end
  return distance
end

def random_permutation(cities)
  perm = Array.new(cities.size){|i| i}
  perm.each_index do |i|
    r = rand(perm.size-i) + i
    perm[r], perm[i] = perm[i], perm[r]
  end
  return perm
end

def initialise_pheromone_matrix(num_cities, naive_score)
  v = num_cities.to_f / naive_score
  return Array.new(num_cities){|i| Array.new(num_cities, v)}
end

def calculate_choices(cities, last_city, exclude, pheromone, c_heur, c_hist)
  choices = []
  cities.each_with_index do |coord, i|
    next if exclude.include?(i)
    prob = {:city=>i}
    prob[:history] = pheromone[last_city][i] ** c_hist
    prob[:distance] = euc_2d(cities[last_city], coord)
    prob[:heuristic] = (1.0/prob[:distance]) ** c_heur
    prob[:prob] = prob[:history] * prob[:heuristic]
    choices << prob
  end
  choices
end

def select_next_city(choices)
  sum = choices.inject(0.0){|sum,element| sum + element[:prob]}
  return choices[rand(choices.size)][:city] if sum == 0.0
  v = rand()
  choices.each_with_index do |choice, i|
    v -= (choice[:prob]/sum)
    return choice[:city] if v <= 0.0
  end
  return choices.last[:city]
end

```

```ruby
def stepwise_const(cities, phero, c_heur, c_hist)
  perm = []
  perm << rand(cities.size)
  begin
    choices = calculate_choices(cities,perm.last,perm,phero,c_heur,c_hist)
    next_city = select_next_city(choices)
    perm << next_city
  end until perm.size == cities.size
  return perm
end

def decay_pheromone(pheromone, decay_factor)
  pheromone.each do |array|
    array.each_with_index do |p, i|
      array[i] = (1.0 - decay_factor) * p
    end
  end
end

def update_pheromone(pheromone, solutions)
  solutions.each do |other|
    other[:vector].each_with_index do |x, i|
      y=(i==other[:vector].size-1) ? other[:vector][0] : other[:vector][i+1]
      pheromone[x][y] += (1.0 / other[:cost])
      pheromone[y][x] += (1.0 / other[:cost])
    end
  end
end

def search(cities, max_it, num_ants, decay_factor, c_heur, c_hist)
  best = {:vector=>random_permutation(cities)}
  best[:cost] = cost(best[:vector], cities)
  pheromone = initialise_pheromone_matrix(cities.size, best[:cost])
  max_it.times do |iter|
    solutions = []
    num_ants.times do
      candidate = {}
      candidate[:vector] = stepwise_const(cities, pheromone, c_heur, c_hist)
      candidate[:cost] = cost(candidate[:vector], cities)
      best = candidate if candidate[:cost] < best[:cost]
    end
    decay_pheromone(pheromone, decay_factor)
    update_pheromone(pheromone, solutions)
    puts " > iteration #{(iter+1)}, best=#{best[:cost]}"
  end
  return best
end

if __FILE__ == $0
  # problem configuration
  berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],
   [880,660],[25,230],[525,1000],[580,1175],[650,1130],[1605,620],
   [1220,580],[1465,200],[1530,5],[845,680],[725,370],[145,665],
   [415,635],[510,875],[560,365],[300,465],[520,585],[480,415],
   [835,625],[975,580],[1215,245],[1320,315],[1250,400],[660,180],
   [410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
    [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],
    [95,260],[875,920],[700,500],[555,815],[830,485],[1170,65],
    [830,610],[605,625],[595,360],[1340,725],[1740,245]]
  # algorithm configuration
  max_it = 50
  num_ants = 30
  decay_factor = 0.6
  c_heur = 2.5
  c_hist = 1.0
  # execute the algorithm
  best = search(berlin52, max_it, num_ants, decay_factor, c_heur, c_hist)
  puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
end
```

Listing 6.2: Ant System in Ruby

## 6.3.8 References

### Primary Sources

The Ant System was described by Dorigo, Maniezzo, and Colorni in an early technical report as a class of algorithms and was applied to a number of standard combinatorial optimization algorithms [4]. A series of technical reports at this time investigated the class of algorithms called Ant System and the specific implementation called Ant Cycle. This effort contributed to Dorigo's PhD thesis published in Italian [2]. The seminal publication into the investigation of Ant System (with the implementation still referred to as Ant Cycle) was by Dorigo in 1996 [3].

### Learn More

The seminal book on Ant Colony Optimization in general with a detailed treatment of Ant system is "Ant colony optimization" by Dorigo and Stützle [5]. An earlier book "Swarm intelligence: from natural to artificial systems" by Bonabeau, Dorigo, and Theraulaz also provides an introduction to Swarm Intelligence with a detailed treatment of Ant System [1].

## 6.3.9 Bibliography

[1] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press US, 1999.

[2] M. Dorigo. *Optimization, Learning and Natural Algorithms (in Italian).* PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992.

[3] M. Dorigo. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and CyberneticsPart B*, 1:1–13, 1996.

[4] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy. Technical report, ipartimento di Elettronica, Politecnico di Milano, Milano, Italy, 1991.

[5] M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, 2004.

## 6.4 Ant Colony System

*Ant Colony System, ACS, Ant-Q.*

### 6.4.1 Taxonomy

The Ant Colony System algorithm is an example of an Ant Colony Optimization method from the field of Swarm Intelligence, Metaheuristics and Computational Intelligence. Ant Colony System is an extension to the Ant System algorithm and is related to other Ant Colony Optimization methods such as Elite Ant System, and Rank-based Ant System.

### 6.4.2 Inspiration

The Ant Colony System algorithm is inspired by the foraging behavior of ants, specifically the pheromone communication between ants regarding a good path between the colony and a food source in an environment. This mechanism is called stigmergy.

### 6.4.3 Metaphor

Ants initially wander randomly around their environment. Once food is located an ant will begin laying down pheromone in the environment. Numerous trips between the food and the colony are performed and if the same route is followed that leads to food then additional pheromone is laid down. Pheromone decays in the environment, so that older paths are less likely to be followed. Other ants may discover the same path to the food and in turn may follow it and also lay down pheromone. A positive feedback process routes more and more ants to productive paths that are in turn further refined through use.

### 6.4.4 Strategy

The objective of the strategy is to exploit historic and heuristic information to construct candidate solutions and fold the information learned from constructing solutions into the history. Solutions are constructed one discrete piece at a time in a probabilistic step-wise manner. The probability of selecting a component is determined by the heuristic contribution of the component to the overall cost of the solution and the quality of solutions from which the component has historically known to have been included. History is updated proportional to the quality of the best known solution and is decreased proportional to the usage if discrete solution components.

### 6.4.5 Procedure

Algorithm 6.4.1 provides a pseudocode listing of the main Ant Colony System algorithm for minimizing a cost function. The probabilistic step-wise construction of solution makes use of both history (pheromone) and problem-specific heuristic information to incrementally construct a solution piece-by-piece. Each component can only be selected if it has not already been chosen (for most combinatorial problems), and for those components that can be selected from given the current component $i$, their probability for selection is defined as:

$$P_{i,j} \leftarrow \frac{\tau_{i,j}^{\alpha} \times \eta_{i,j}^{\beta}}{\sum_{k=1}^{c} \tau_{i,k}^{\alpha} \times \eta_{i,k}^{\beta}} \tag{6.4}$$

where $\eta_{i,j}$ is the maximizing contribution to the overall score of selecting the component (such as $\frac{1.0}{distance_{i,j}}$ for the Traveling Salesman Problem), $\beta$ is the heuristic coefficient (commonly fixed at 1.0), $\tau_{i,j}$ is the pheromone value for the component, $\alpha$ is the history coefficient, and $c$ is the set of usable components. A greediness factor ($q0$) is used to influence when to use the above probabilistic component selection and when to greedily select the best possible component.

A local pheromone update is performed for each solution that is constructed to dissuade following solutions to use the same components in the same order, as follows:

$$\tau_{i,j} \leftarrow (1-\sigma) \times \tau_{i,j} + \sigma \times \tau_{i,j}^{0} \tag{6.5}$$

where $\tau_{i,j}$ represents the pheromone for the component (graph edge) $(i,j)$, $\sigma$ is the local pheromone factor, and $\tau_{i,j}^{0}$ is the initial pheromone value.

At the end of each iteration, the pheromone is updated and decayed using the best candidate solution found thus far (or the best candidate solution found for the iteration), as follows:

$$\tau_{i,j} \leftarrow (1-\rho) \times \tau_{i,j} + \rho \times \Delta\tau i,j \tag{6.6}$$

where $\tau_{i,j}$ represents the pheromone for the component (graph edge) $(i,j)$, $\rho$ is the decay factor, and $\Delta\tau i,j$ is the maximizing solution cost for the best solution found so far if the component $ij$ is used in the globally best known solution, otherwise it is 0.

### 6.4.6 Heuristics

- The Ant Colony System algorithm was designed for use with combinatorial problems such as the TSP, knapsack problem, quadratic assignment problems, graph coloring problems and many others.

**Algorithm 6.4.1**: Pseudocode for Ant Colony System.

```
   Input: ProblemSize, $Population_{size}$, $m$, $\rho$, $\beta$, $\sigma$, $q0$
   Output: $P_{best}$
1  $P_{best}$ ← CreateHeuristicSolution(ProblemSize);
2  $Pbest_{cost}$ ← Cost($S_h$);
3  $Pheromone_{init}$ ← $\frac{1.0}{\text{ProblemSize} \times Pbest_{cost}}$;
4  Pheromone ← InitializePheromone($Pheromone_{init}$);
5  while ¬StopCondition() do
6      for $i = 1$ to $m$ do
7          $S_i$ ← ConstructSolution(Pheromone, ProblemSize, $\beta$, $q0$);
8          $Si_{cost}$ ← Cost($S_i$);
9          if $Si_{cost} \leq Pbest_{cost}$ then
10             $Pbest_{cost}$ ← $Si_{cost}$;
11             $P_{best}$ ← $S_i$;
12         end
13         LocalUpdateAndDecayPheromone(Pheromone, $S_i$, $Si_{cost}$, $\sigma$);
14     end
15     GlobalUpdateAndDecayPheromone(Pheromone, $P_{best}$, $Pbest_{cost}$,
       $\rho$);
16 end
17 return $P_{best}$;
```

- The local pheromone (history) coefficient ($\sigma$) controls the amount of contribution history plays in a components probability of selection and is commonly set to 0.1.

- The heuristic coefficient ($\beta$) controls the amount of contribution problem-specific heuristic information plays in a components probability of selection and is commonly between 2 and 5, such as 2.5.

- The decay factor ($\rho$) controls the rate at which historic information is lost and is commonly set to 0.1.

- The greediness factor ($q0$) is commonly set to 0.9.

- The total number of ants ($m$) is commonly set low, such as 10.

### 6.4.7 Code Listing

Listing 6.3 provides an example of the Ant Colony System algorithm implemented in the Ruby Programming Language. The algorithm is applied to the Berlin52 instance of the Traveling Salesman Problem (TSP), taken from the TSPLIB. The problem seeks a permutation of the order to visit cities (called a tour) that minimized the total distance traveled. The optimal tour distance for Berlin52 instance is 7542 units. Some extensions to the algorithm implementation for speed improvements may consider pre-calculating a distance matrix for all the cities in the problem, and pre-computing a probability matrix for choices during the probabilistic step-wise construction of tours.

```ruby
def euc_2d(c1, c2)
  Math.sqrt((c1[0] - c2[0])**2.0 + (c1[1] - c2[1])**2.0).round
end

def cost(permutation, cities)
  distance =0
  permutation.each_with_index do |c1, i|
    c2 = (i==permutation.size-1) ? permutation[0] : permutation[i+1]
    distance += euc_2d(cities[c1], cities[c2])
  end
  return distance
end

def random_permutation(cities)
  perm = Array.new(cities.size){|i| i}
  perm.each_index do |i|
    r = rand(perm.size-i) + i
    perm[r], perm[i] = perm[i], perm[r]
  end
  return perm
end

def initialise_pheromone_matrix(num_cities, init_pher)
  return Array.new(num_cities){|i| Array.new(num_cities, init_pher)}
end

def calculate_choices(cities, last_city, exclude, pheromone, c_heur, c_hist)
  choices = []
  cities.each_with_index do |coord, i|
    next if exclude.include?(i)
    prob = {:city=>i}
    prob[:history] = pheromone[last_city][i] ** c_hist
    prob[:distance] = euc_2d(cities[last_city], coord)
    prob[:heuristic] = (1.0/prob[:distance]) ** c_heur
    prob[:prob] = prob[:history] * prob[:heuristic]
    choices << prob
  end
  return choices
end

def prob_select(choices)
  sum = choices.inject(0.0){|sum,element| sum + element[:prob]}
  return choices[rand(choices.size)][:city] if sum == 0.0
  v = rand()
  choices.each_with_index do |choice, i|
    v -= (choice[:prob]/sum)
    return choice[:city] if v <= 0.0
  end
  return choices.last[:city]
```

```ruby
end

def greedy_select(choices)
  return choices.max{|a,b| a[:prob]<=>b[:prob]}[:city]
end

def stepwise_const(cities, phero, c_heur, c_greed)
  perm = []
  perm << rand(cities.size)
  begin
    choices = calculate_choices(cities, perm.last, perm, phero, c_heur, 1.0)
    greedy = rand() <= c_greed
    next_city = (greedy) ? greedy_select(choices) : prob_select(choices)
    perm << next_city
  end until perm.size == cities.size
  return perm
end

def global_update_pheromone(phero, cand, decay)
  cand[:vector].each_with_index do |x, i|
    y = (i==cand[:vector].size-1) ? cand[:vector][0] : cand[:vector][i+1]
    value = ((1.0-decay)*phero[x][y]) + (decay*(1.0/cand[:cost]))
    phero[x][y] = value
    phero[y][x] = value
  end
end

def local_update_pheromone(pheromone, cand, c_local_phero, init_phero)
  cand[:vector].each_with_index do |x, i|
    y = (i==cand[:vector].size-1) ? cand[:vector][0] : cand[:vector][i+1]
    value = ((1.0-c_local_phero)*pheromone[x][y])+(c_local_phero*init_phero)
    pheromone[x][y] = value
    pheromone[y][x] = value
  end
end

def search(cities, max_it, num_ants, decay, c_heur, c_local_phero, c_greed)
  best = {:vector=>random_permutation(cities)}
  best[:cost] = cost(best[:vector], cities)
  init_pheromone = 1.0 / (cities.size.to_f * best[:cost])
  pheromone = initialise_pheromone_matrix(cities.size, init_pheromone)
  max_it.times do |iter|
    solutions = []
    num_ants.times do
      cand = {}
      cand[:vector] = stepwise_const(cities, pheromone, c_heur, c_greed)
      cand[:cost] = cost(cand[:vector], cities)
      best = cand if cand[:cost] < best[:cost]
      local_update_pheromone(pheromone, cand, c_local_phero, init_pheromone)
    end
    global_update_pheromone(pheromone, best, decay)
    puts " > iteration #{(iter+1)}, best=#{best[:cost]}"
  end
  return best
end

```

```
if __FILE__ == $0
  # problem configuration
  berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],
   [880,660],[25,230],[525,1000],[580,1175],[650,1130],[1605,620],
   [1220,580],[1465,200],[1530,5],[845,680],[725,370],[145,665],
   [415,635],[510,875],[560,365],[300,465],[520,585],[480,415],
   [835,625],[975,580],[1215,245],[1320,315],[1250,400],[660,180],
   [410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
   [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],
   [95,260],[875,920],[700,500],[555,815],[830,485],[1170,65],
   [830,610],[605,625],[595,360],[1340,725],[1740,245]]
  # algorithm configuration
  max_it = 100
  num_ants = 10
  decay = 0.1
  c_heur = 2.5
  c_local_phero = 0.1
  c_greed = 0.9
  # execute the algorithm
  best = search(berlin52, max_it, num_ants, decay, c_heur, c_local_phero,
       c_greed)
  puts "Done. Best Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
end
```

Listing 6.3: Ant Colony System in Ruby

### 6.4.8 References

#### Primary Sources

The algorithm was initially investigated by Dorigo and Gambardella under the name Ant-Q [2, 6]. It was renamed Ant Colony System and further investigated first in a technical report by Dorigo and Gambardella [4], and later published [3].

#### Learn More

The seminal book on Ant Colony Optimization in general with a detailed treatment of Ant Colony System is "Ant colony optimization" by Dorigo and Stützle [5]. An earlier book "Swarm intelligence: from natural to artificial systems" by Bonabeau, Dorigo, and Theraulaz also provides an introduction to Swarm Intelligence with a detailed treatment of Ant Colony System [1].

### 6.4.9 Bibliography

[1] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press US, 1999.

[2] M. Dorigo and L. M. Gambardella. A study of some properties of ant-q. In H-M. Voigt, W. Ebeling, I. Rechenberg, and H-P. Schwefel, editors,

*Proceedings of PPSN IVFourth International Conference on Parallel Problem Solving From Nature*, pages 656–665. Springer-Verlag, 1996.

[3] M. Dorigo and L. M. Gambardella. Ant colony system : A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[4] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problems. Technical Report TR/IRIDIA/1996-5, IRIDIA, Université Libre de Bruxelles, 1997.

[5] M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, 2004.

[6] L. Gambardella and M. Dorigo. Ant–Q: A reinforcement learning approach to the traveling salesman problems. In A. Prieditis and S. Russell, editors, *Proceedings of ML-95, Twelfth International Conference on Machine Learning*, pages 252–260. Morgan Kaufmann, 1995.

## 6.5 Bees Algorithm

*Bees Algorithm, BA.*

### 6.5.1 Taxonomy

The Bees Algorithm beings to Bee Inspired Algorithms and the field of Swarm Intelligence, and more broadly the fields of Computational Intelligence and Metaheuristics. The Bees Algorithm is related to other Bee Inspired Algorithms, such as Bee Colony Optimization, and other Swarm Intelligence algorithms such as Ant Colony Optimization and Particle Swarm Optimization.

### 6.5.2 Inspiration

The Bees Algorithm is inspired by the foraging behavior of honey bees. Honey bees collect nectar from vast areas around their hive (more than 10 kilometers). Bee Colonies have been observed to send bees to collect nectar from flower patches relative to the amount of food available at each patch. Bees communicate with each other at the hive via a waggle dance that informs other bees in the hive as to the direction, distance, and quality rating of food sources.

### 6.5.3 Metaphor

Honey bees collect nectar from flower patches as a food source for the hive. The hive sends out scout's that locate patches of flowers, who then return to the hive and inform other bees about the fitness and location of a food source via a waggle dance. The scout returns to the flower patch with follower bees. A small number of scouts continue to search for new patches, while bees returning from flower patches continue to communicate the quality of the patch.

### 6.5.4 Strategy

The information processing objective of the algorithm is to locate and explore good sites within a problem search space. Scouts are sent out to randomly sample the problem space and locate good sites. The good sites are exploited via the application of a local search, where a small number of good sites are explored more than the others. Good sites are continually exploited, although many scouts are sent out each iteration always in search of additional good sites.

## 6.5.5 Procedure

Algorithm 6.5.1 provides a pseudocode listing of the Bees Algorithm for minimizing a cost function.

**Algorithm 6.5.1**: Pseudocode for the Bees Algorithm.

```
Input: $Problem_{size}$, $Bees_{num}$, $Sites_{num}$, $EliteSites_{num}$,
       $PatchSize_{init}$, $EliteBees_{num}$, $OtherBees_{num}$
Output: $Bee_{best}$
Population ← InitializePopulation($Bees_{num}$, $Problem_{size}$);
while ¬StopCondition() do
    EvaluatePopulation(Population);
    $Bee_{best}$ ← GetBestSolution(Population);
    NextGeneration ← ∅;
    $Patch_{size}$ ← ( $PatchSize_{init}$ × $PatchDecrease_{factor}$);
    $Sites_{best}$ ← SelectBestSites(Population, $Sites_{num}$);
    foreach $Site_i \in Sites_{best}$ do
        $RecruitedBees_{num}$ ← ∅;
        if $i < EliteSites_{num}$ then
            $RecruitedBees_{num}$ ← $EliteBees_{num}$;
        else
            $RecruitedBees_{num}$ ← $OtherBees_{num}$;
        end
        Neighborhood ← ∅;
        for $j$ to $RecruitedBees_{num}$ do
            Neighborhood ← CreateNeighborhoodBee($Site_i$,
            $Patch_{size}$);
        end
        NextGeneration ← GetBestSolution(Neighborhood);
    end
    $RemainingBees_{num}$ ← ($Bees_{num}$- $Sites_{num}$);
    for $j$ to $RemainingBees_{num}$ do
        NextGeneration ← CreateRandomBee();
    end
    Population ← NextGeneration;
end
return $Bee_{best}$;
```

## 6.5.6 Heuristics

- The Bees Algorithm was developed to be used with continuous and combinatorial function optimization problems.
- The $Patch_{size}$ variable is used as the neighborhood size. For example,

in a continuous function optimization problem, each dimension of a site would be sampled as $x_i \pm (rand() \times Patch_{size})$.

- The $Patch_{size}$ variable is decreased each iteration, typically by a constant amount (such as 0.95).
- The number of elite sites ($EliteSites_{num}$) must be $<$ the number of sites ($Sites_{num}$), and the number of elite bees ($EliteBees_{num}$) is traditionally $<$ the number of other bees ($OtherBees_{num}$).

## 6.5.7 Code Listing

Listing 6.4 provides an example of the Bees Algorithm 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 an implementation of the Bees Algorithm as described in the seminal paper [2]. A fixed patch size decrease factor of 0.95 was applied each iteration.

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

def random_vector(minmax)
  return Array.new(minmax.size) do |i|
    minmax[i][0] + ((minmax[i][1] - minmax[i][0]) * rand())
  end
end

def create_random_bee(search_space)
  return {:vector=>random_vector(search_space)}
end

def create_neigh_bee(site, patch_size, search_space)
  vector = []
  site.each_with_index do |v,i|
    v = (rand()<0.5) ? v+rand()*patch_size : v-rand()*patch_size
    v = search_space[i][0] if v < search_space[i][0]
    v = search_space[i][1] if v > search_space[i][1]
    vector << v
  end
  bee = {}
  bee[:vector] = vector
  return bee
end

def search_neigh(parent, neigh_size, patch_size, search_space)
  neigh = []
  neigh_size.times do
    neigh << create_neigh_bee(parent[:vector], patch_size, search_space)
  end
  neigh.each{|bee| bee[:fitness] = objective_function(bee[:vector])}
  return neigh.sort{|x,y| x[:fitness]<=>y[:fitness]}.first
end

def create_scout_bees(search_space, num_scouts)
  return Array.new(num_scouts) do
    create_random_bee(search_space)
  end
end

def search(max_gens, search_space, num_bees, num_sites, elite_sites,
     patch_size, e_bees, o_bees)
  best = nil
  pop = Array.new(num_bees){ create_random_bee(search_space) }
  max_gens.times do |gen|
    pop.each{|bee| bee[:fitness] = objective_function(bee[:vector])}
    pop.sort!{|x,y| x[:fitness]<=>y[:fitness]}
    best = pop.first if best.nil? or pop.first[:fitness] < best[:fitness]
    next_gen = []
    pop[0...num_sites].each_with_index do |parent, i|
      neigh_size = (i<elite_sites) ? e_bees : o_bees
      next_gen << search_neigh(parent, neigh_size, patch_size, search_space)
    end
    scouts = create_scout_bees(search_space, (num_bees-num_sites))
    pop = next_gen + scouts
    patch_size = patch_size * 0.95
    puts " > it=#{gen+1}, patch_size=#{patch_size}, f=#{best[:fitness]}"
  end
  return best
end

if __FILE__ == $0
  # problem configuration
  problem_size = 3
  search_space = Array.new(problem_size) {|i| [-5, 5]}
  # algorithm configuration
  max_gens = 500
  num_bees = 45
  num_sites = 3
  elite_sites = 1
  patch_size = 3.0
  e_bees = 7
  o_bees = 2
  # execute the algorithm
  best = search(max_gens, search_space, num_bees, num_sites, elite_sites,
       patch_size, e_bees, o_bees)
  puts "done! Solution: f=#{best[:fitness]}, s=#{best[:vector].inspect}"
end
```

Listing 6.4: Bees Algorithm in Ruby

### 6.5.8 References

#### Primary Sources

The Bees Algorithm was proposed by Pham et al. in a technical report in 2005 [3], and later published [2]. In this work, the algorithm was applied to standard instances of continuous function optimization problems.

#### Learn More

The majority of the work on the algorithm has concerned its application to various problem domains. The following is a selection of popular application papers: the optimization of linear antenna arrays by Guney and Onay [1], the optimization of codebook vectors in the Learning Vector Quantization algorithm for classification by Pham et al. [5], optimization of neural networks for classification by Pham et al. [6], and the optimization of clustering methods by Pham et al. [4].

### 6.5.9 Bibliography

[1] K. Guney and M. Onay. Amplitude-only pattern nulling of linear antenna arrays with the use of bees algorithm. *Progress In Electromagnetics Research*, 70:21–36, 2007.

[2] D. T. Pham, Ghanbarzadeh A., Koc E., Otri S., Rahim S., and M.Zaidi. The bees algorithm - a novel tool for complex optimisation problems. In *Proceedings of IPROMS 2006 Conference*, pages 454–461, 2006.

[3] D. T. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim, and M. Zaidi. The bees algorithm. Technical report, Manufacturing Engineering Centre, Cardiff University, 2005.

[4] D. T. Pham, S. Otri, A. A. Afify, M. Mahmuddin, and H. Al-Jabbouli. Data clustering using the bees algorithm. In *Proc 40th CIRP International Manufacturing Systems Seminar*, 2007.

[5] D. T. Pham, S. Otri, A. Ghanbarzadeh, and E. Koc. Application of the bees algorithm to the training of learning vector quantisation networks for control chart pattern recognition. In *Proceedings of Information and Communication Technologies (ICTTA'06)*, pages 1624–1629, 2006.

[6] D. T. Pham, A. J. Soroka, A. Ghanbarzadeh, E. Koc, S. Otri, and M. Packianather. Optimising neural networks for identification of wood defects using the bees algorithm. In *Proceedings of the 2006 IEEE International Conference on Industrial Informatics*, 2006.

## 6.6 Bacterial Foraging Optimization Algorithm

*Bacterial Foraging Optimization Algorithm, BFOA, Bacterial Foraging Optimization, BFO.*

### 6.6.1 Taxonomy

The Bacterial Foraging Optimization Algorithm belongs to the field of Bacteria Optimization Algorithms and Swarm Optimization, and more broadly to the fields of Computational Intelligence and Metaheuristics. It is related to other Bacteria Optimization Algorithms such as the Bacteria Chemotaxis Algorithm [3], and other Swarm Intelligence algorithms such as Ant Colony Optimization and Particle Swarm Optimization. There have been many extensions of the approach that attempt to hybridize the algorithm with other Computational Intelligence algorithms and Metaheuristics such as Particle Swarm Optimization, Genetic Algorithm, and Tabu Search.

### 6.6.2 Inspiration

The Bacterial Foraging Optimization Algorithm is inspired by the group foraging behavior of bacteria such as E.coli and M.xanthus. Specifically, the BFOA is inspired by the chemotaxis behavior of bacteria that will perceive chemical gradients in the environment (such as nutrients) and move toward or away from specific signals.

### 6.6.3 Metaphor

Bacteria perceive the direction to food based on the gradients of chemicals in their environment. Similarly, bacteria secrete attracting and repelling chemicals into the environment and can perceive each other in a similar way. Using locomotion mechanisms (such as flagella) bacteria can move around in their environment, sometimes moving chaotically (tumbling and spinning), and other times moving in a directed manner that may be referred to as swimming. Bacterial cells are treated like agents in an environment, using their perception of food and other cells as motivation to move, and stochastic tumbling and swimming like movement to re-locate. Depending on the cell-cell interactions, cells may swarm a food source, and/or may aggressively repel or ignore each other.

### 6.6.4 Strategy

The information processing strategy of the algorithm is to allow cells to stochastically and collectively swarm toward optima. This is achieved through a series of three processes on a population of simulated cells: 1) ‘Chemotaxis’ where the cost of cells is derated by the proximity to other cells and cells move along the manipulated cost surface one at a time (the majority of the work of the algorithm), 2) 'Reproduction' where only those cells that performed well over their lifetime may contribute to the next generation, and 3) 'Elimination-dispersal' where cells are discarded and new random samples are inserted with a low probability.

### 6.6.5 Procedure

Algorithm 6.6.1 provides a pseudocode listing of the Bacterial Foraging Optimization Algorithm for minimizing a cost function. Algorithm 6.6.2 provides the pseudocode listing for the chemotaxis and swing behaviour of the BFOA algorithm. A bacteria cost is derated by its interaction with other cells. This interaction function ($g()$) is calculated as follows:

$$g(cell_k) = \sum_{i=1}^{S}\Bigg[-d_{attr}\times exp\bigg(-w_{attr}\times\sum_{m=1}^{P}(cell_m^k - other_m^i)^2\bigg)\Bigg] + \sum_{i=1}^{S}\Bigg[h_{repel}\times exp\bigg(-w_{repel}\times\sum_{m=1}^{P}cell_m^k - other_m^i)^2\bigg)\Bigg]$$

where $cell_k$ is a given cell, $d_{attr}$ and $w_{attr}$ are attraction coefficients, $h_{repel}$ and $w_{repel}$ are repulsion coefficients, $S$ is the number of cells in the population, $P$ is the number of dimensions on a given cells position vector.

The remaining parameters of the algorithm are as follows $Cells_{num}$ is the number of cells maintained in the population, $N_{ed}$ is the number of elimination-dispersal steps, $N_{re}$ is the number of reproduction steps, $N_c$ is the number of chemotaxis steps, $N_s$ is the number of swim steps for a given cell, $Step_{size}$ is a random direction vector with the same number of dimensions as the problem space, and each value $\in[-1,1]$, and $P_{ed}$ is the probability of a cell being subjected to elimination and dispersal.

### 6.6.6 Heuristics

- The algorithm was designed for application to continuous function optimization problem domains.
- Given the loops in the algorithm, it can be configured numerous ways to elicit different search behavior. It is common to have a large number of chemotaxis iterations, and small numbers of the other iterations.
- The default coefficients for swarming behavior (cell-cell interactions) are as follows $d_{attract}=0.1$, $w_{attract}=0.2$, $h_{repellant}=d_{attract}$, and $w_{repellant}=10$.
- The step size is commonly a small fraction of the search space, such as 0.1.

**Algorithm 6.6.1**: Pseudocode for the BFOA.

```
Input: $Problem_{size}$, $Cells_{num}$, $N_{ed}$, $N_{re}$, $N_c$, $N_s$, $Step_{size}$, $d_{attract}$,
       $w_{attract}$, $h_{repellant}$, $w_{repellant}$, $P_{ed}$
Output: $Cell_{best}$
1  Population ← InitializePopulation($Cells_{num}$, $Problem_{size}$);
2  for $l = 0$ to $N_{ed}$ do
3      for $k = 0$ to $N_{re}$ do
4          for $j = 0$ to $N_c$ do
5              ChemotaxisAndSwim(Population, $Problem_{size}$, $Cells_{num}$,
               $N_s$, $Step_{size}$, $d_{attract}$, $w_{attract}$, $h_{repellant}$, $w_{repellant}$);
6              foreach Cell ∈ Population do
7                  if Cost(Cell) $\leq$ Cost($Cell_{best}$) then
8                      $Cell_{best}$ ← Cell;
9                  end
10             end
11         end
12         SortByCellHealth(Population);
13         Selected ← SelectByCellHealth(Population, $\frac{Cells_{num}}{2}$);
14         Population ← Selected;
15         Population ← Selected;
16     end
17     foreach Cell ∈ Population do
18         if Rand() $\leq P_{ed}$ then
19             Cell ← CreateCellAtRandomLocation();
20         end
21     end
22 end
23 return $Cell_{best}$;
```

- During reproduction, typically half the population with a low health metric are discarded, and two copies of each member from the first (high-health) half of the population are retained.
- The probability of elimination and dispersal ($p_{ed}$) is commonly set quite large, such as 0.25.

## 6.6.7 Code Listing

Listing 6.5 provides an example of the Bacterial Foraging Optimization Algorithm 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 $n = 2$. The optimal solution for this basin function is $(v_0, \ldots, v_{n-1}) = 0.0$. The algorithm

**Algorithm 6.6.2**: Pseudocode for the `ChemotaxisAndSwim` function.

```
Input: Population, $Problem_{size}$, $Cells_{num}$, $N_s$, $Step_{size}$, $d_{attract}$,
       $w_{attract}$, $h_{repellant}$, $w_{repellant}$
1  foreach Cell ∈ Population do
2      $Cell_{fitness}$ ← Cost(Cell) + Interaction(Cell, Population,
       $d_{attract}$, $w_{attract}$, $h_{repellant}$, $w_{repellant}$);
3      $Cell_{health} \leftarrow Cell_{fitness}$;
4      $Cell' \leftarrow \emptyset$;
5      for $i = 0$ to $N_s$ do
6          RandomStepDirection ← CreateStep($Problem_{size}$);
7          $Cell'$ ← TakeStep(RandomStepDirection, $Step_{size}$);
8          $Cell'_{fitness}$ ← Cost($Cell'$) + Interaction($Cell'$, Population,
           $d_{attract}$, $w_{attract}$, $h_{repellant}$, $w_{repellant}$);
9          if $Cell'_{fitness} > Cell_{fitness}$ then
10             $i \leftarrow N_s$;
11         else
12             Cell ← $Cell'$;
13             $Cell_{health} \leftarrow Cell_{health} + Cell'_{fitness}$;
14         end
15     end
16 end
```

is an implementation based on the description on the seminal work [4]. The parameters for cell-cell interactions (attraction and repulsion) were taken from the paper, and the various loop parameters were taken from the 'Swarming Effects' example.

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

def random_vector(minmax)
  return Array.new(minmax.size) do |i|
    minmax[i][0] + ((minmax[i][1] - minmax[i][0]) * rand())
  end
end

def generate_random_direction(problem_size)
  bounds = Array.new(problem_size){[-1.0,1.0]}
  return random_vector(bounds)
end

def compute_cell_interaction(cell, cells, d, w)
  sum = 0.0
  cells.each do |other|
    diff = 0.0
    cell[:vector].each_index do |i|
      diff += (cell[:vector][i] - other[:vector][i])**2.0
    end
    sum += d * Math.exp(w * diff)
  end
  return sum
end

def attract_repel(cell, cells, d_attr, w_attr, h_rep, w_rep)
  attract = compute_cell_interaction(cell, cells, -d_attr, -w_attr)
  repel = compute_cell_interaction(cell, cells, h_rep, -w_rep)
  return attract + repel
end

def evaluate(cell, cells, d_attr, w_attr, h_rep, w_rep)
  cell[:cost] = objective_function(cell[:vector])
  cell[:inter] = attract_repel(cell, cells, d_attr, w_attr, h_rep, w_rep)
  cell[:fitness] = cell[:cost] + cell[:inter]
end

def tumble_cell(search_space, cell, step_size)
  step = generate_random_direction(search_space.size)
  vector = Array.new(search_space.size)
  vector.each_index do |i|
    vector[i] = cell[:vector][i] + step_size * step[i]
    vector[i] = search_space[i][0] if vector[i] < search_space[i][0]
    vector[i] = search_space[i][1] if vector[i] > search_space[i][1]
  end
  return {:vector=>vector}
end

def chemotaxis(cells, search_space, chem_steps, swim_length, step_size,
     d_attr, w_attr, h_rep, w_rep)
  best = nil
  chem_steps.times do |j|
    moved_cells = []
    cells.each_with_index do |cell, i|
      sum_nutrients = 0.0
      evaluate(cell, cells, d_attr, w_attr, h_rep, w_rep)
      best = cell if best.nil? or cell[:cost] < best[:cost]
      sum_nutrients += cell[:fitness]
      swim_length.times do |m|
        new_cell = tumble_cell(search_space, cell, step_size)
        evaluate(new_cell, cells, d_attr, w_attr, h_rep, w_rep)
        best = cell if cell[:cost] < best[:cost]
        break if new_cell[:fitness] > cell[:fitness]
        cell = new_cell
        sum_nutrients += cell[:fitness]
      end
      cell[:sum_nutrients] = sum_nutrients
      moved_cells << cell
    end
    puts " >> chemo=#{j}, f=#{best[:fitness]}, cost=#{best[:cost]}"
    cells = moved_cells
  end
  return [best, cells]
end
```

```ruby

def search(search_space, pop_size, elim_disp_steps, repro_steps,
     chem_steps, swim_length, step_size, d_attr, w_attr, h_rep, w_rep,
     p_eliminate)
  cells = Array.new(pop_size) { {:vector=>random_vector(search_space)} }
  best = nil
  elim_disp_steps.times do |l|
    repro_steps.times do |k|
      c_best, cells = chemotaxis(cells, search_space, chem_steps,
           swim_length, step_size, d_attr, w_attr, h_rep, w_rep)
      best = c_best if best.nil? or c_best[:cost] < best[:cost]
      puts " > best fitness=#{best[:fitness]}, cost=#{best[:cost]}"
      cells.sort{|x,y| x[:sum_nutrients]<=>y[:sum_nutrients]}
      cells = cells.first(pop_size/2) + cells.first(pop_size/2)
    end
    cells.each do |cell|
      if rand() <= p_eliminate
        cell[:vector] = random_vector(search_space)
      end
    end
  end
  return best
end

if __FILE__ == $0
  # problem configuration
  problem_size = 2
  search_space = Array.new(problem_size) {|i| [-5, 5]}
  # algorithm configuration
  pop_size = 50
  step_size = 0.1 # Ci
  elim_disp_steps = 1 # Ned
  repro_steps = 4 # Nre
  chem_steps = 70 # Nc
  swim_length = 4 # Ns
  p_eliminate = 0.25 # Ped
  d_attr = 0.1
  w_attr = 0.2
  h_rep = d_attr
  w_rep = 10
  # execute the algorithm
  best = search(search_space, pop_size, elim_disp_steps, repro_steps,
       chem_steps, swim_length, step_size, d_attr, w_attr, h_rep, w_rep,
       p_eliminate)
  puts "done! Solution: c=#{best[:cost]}, v=#{best[:vector].inspect}"
end
```

Listing 6.5: Bacterial Foraging Optimization Algorithm in Ruby

## 6.6.8 References

### Primary Sources

Early work by Liu and Passino considered models of chemotaxis as optimization for both E.coli and M.xanthus which were applied to continuous function optimization [2]. This work was consolidated by Passino who presented the Bacterial Foraging Optimization Algorithm that included a detailed presentation of the algorithm, heuristics for configuration, and demonstration applications and behavior dynamics [4].

**Learn More**

A detailed summary of social foraging and the BFOA is provided in the book by Passino [5]. Passino provides a follow-up review of the background models of chemotaxis as optimization and describes the equations of the Bacterial Foraging Optimization Algorithm in detail in a Journal article [6]. Das et al. present the algorithm and its inspiration, and go on to provide an in depth analysis the dynamics of chemotaxis using simplified mathematical models [1].

### 6.6.9 Bibliography

[1] S. Das, A. Biswas, S. Dasgupta, and A. Abraham. *Foundations of Computational Intelligence Volume 3: Global Optimization*, chapter Bacterial Foraging Optimization Algorithm: Theoretical Foundations, Analysis, and Applications, pages 23–55. Springer, 2009.

[2] Y. Liu and K. M. Passino. Biomimicry of social foraging bacteria for distributed optimization: Models, principles, and emergent behaviors. *Journal of Optimization Theory and Applications*, 115(3):603–628, 2002.

[3] S. D. Müller, J. Marchetto, S. Airaghi, and P. Koumoutsakos. Optimization based on bacterial chemotaxis. *IEEE Transactions on Evolutionary Computation*, 6(1):16–29, 2002.

[4] K. M. Passino. Biomimicry of bacterial foraging for distributed optimization and control. *IEEE Control Systems Magazine*, 22(3):52–67, 2002.

[5] K. M. Passino. *Biomimicry for Optimization, Control, and Automation*, chapter Part V: Foraging. Springer, 2005.

[6] K. M. Passino. Bacterial foraging optimization. *International Journal of Swarm Intelligence Research*, 1(1):1–16, 2010.