Chapter 4

A Tutorial on Meta-Heuristics for Optimization

Shu-Chuan Chu, Chin-Shiuh Shieh, and John F. Roddick

Nature has inspired computing and engineering researchers in many different ways. Natural processes have been emulated through a variety of techniques including genetic algorithms, ant systems and particle swarm optimization, as computational models for optimization. In this chapter, we discuss these meta-heuristics from a practitioner's point of view, emphasizing the fundamental ideas and their implementations. After presenting the underlying philosophy and algorithms, detailed implementations are given, followed by some discussion of alternatives.

1 Introduction

Optimization problems arise from almost every field ranging from academic research to industrial application. In addition to other (arguably more conventional) optimization techniques, meta-heuristics, such as genetic algorithms, particle swarm optimization and ant colony systems, have received increasing attention in recent years for their interesting characteristics and their success in solving problems in a number of realms. In this tutorial, we discuss these meta-heuristics from a practicer's point of view. After a brief explanation of the underlying philosophy and a discussion of the algorithms, detailed implementations in C are given, followed by some implementation notes and possible alternatives.

The discussion is not intended to be a comprehensive review of related topics, but a compact guide for implementation, with which readers can put these meta-heuristics to work and experience their power in timely fashion. The C programming language was adopted because of its portability and availability. The coding style is deliberately made as accessible as possible, so that interesting readers can easily transfer the code into any other programming language as preferred[1].

In general, an optimization problem can be modeled as follows:

$$F(\vec{x}), \vec{x} = (x_1, x_2, \cdots, x_n) \in \mathbf{X}, \tag{1}$$

where $F(\vec{x})$ is the object function subject to optimization, and $\mathbf{X}$ is the domain of independent variables $x_1, x_2, \cdots, x_n$. We are asked to find out certain configuration of $\vec{x} = (x_1, x_2, \cdots, x_n)$ to maximize or minimize the object function $F(\vec{x})$.

The optimization task can be challenging for several reasons, such as high dimensionality of $\vec{x}$, constrains imposed on $\vec{x}$, non-differentiability of $F(\vec{x})$, and the existence of local optima.

2 Genetic Algorithms

Based on long-term observation, Darwin asserted his theory of natural evolution. In the natural world, creatures compete with each other for limited resources. Those individuals that survive in the competition have the opportunity to reproduce and generate descendants. In so doing, any exchange of genes may result in superior or inferior descendants with the process of natural selection eventually filtering out inferior individuals and retain those adapted best to their environment.

[1]The code segments can be downloaded from
`http://kdm.first.flinders.edu.au/IDM/`.

Inspired by Darwin's theory of evolution, Holland (Holland 1975, Goldberg 1989) introduced the genetic algorithm as a powerful computational model for optimization. Genetic algorithms work on a population of potential solutions, in the form of chromosomes, and try to locate a best solution through the process of artificial evolution, which consist of repeated artificial genetic operations, namely evaluation, selection, crossover and mutation.

A multi-modal object function $F_1(x, y)$ as shown in Figure 1 is used to illustrate this. The global optimum is located at approximately $F_1(1.9931, 1.9896) = 4.2947$.

$$F_1(x,y) = \frac{4}{(x-2)^2+(y-2)^2+1} + \frac{3}{(x-2)^2+(y+2)^2+1} \frac{2}{(x+2)^2+(y-2)^2+1}, \quad -5 \le x, y < 5 \qquad (2)$$

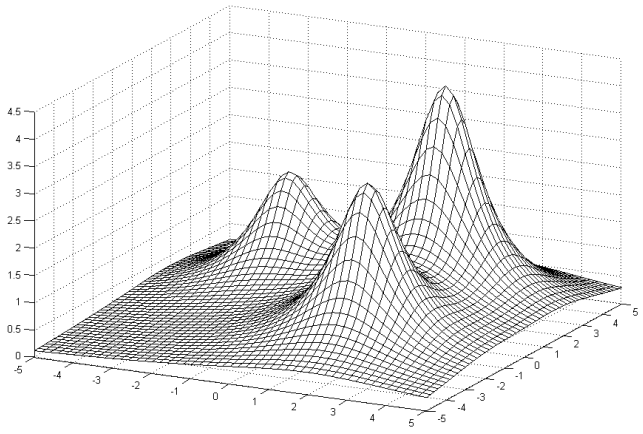


Figure 1. Object function F_1

The first design issue in applying genetic algorithms is to select an adequate coding scheme to represent potential solutions in the search

space in the form of chromosomes. Among other alternatives, such as expression trees for genetic programming (Willis *et al.* 1997) and city index permutation for the travelling salesperson problem, binary string coding is widely used for numerical optimization. Figure 2 gives a typical binary string coding for the test function F_1. Each genotype has 16 bits to encode an independent variable. A decoding function will map the 65536 possible combinations of $b_{15} \cdots b_0$ onto the range $[-5, 5)$ linearly. A chromosome is then formed by cascading genotypes for each variable. With this coding scheme, any 32 bit binary string stands for a legal point in the problem domain.

$$d(b_{15}b_{14} \cdots b_1 b_0) = \frac{\sum_{i=0}^{15} b_i 2^i}{2^{16}} \times 10 - 5 \tag{3}$$

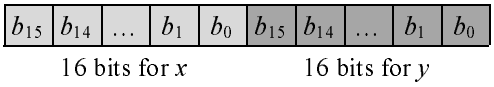


Figure 2. A binary coding scheme for F_1

A second issue is to decide the population size. The choice of population size, N, is a tradeoff between solution quality and computation cost. A larger population size will maintain higher genetic diversity and therefore a higher possibility of locating global optimum, however, at a higher computational cost. The operation of genetic algorithms is outlined as follows:

Step 1 Initialization

Each bit of all N chromosomes in the population is randomly set to 0 or 1. This operation in effect spreads chromosomes randomly into the problem domains. Whenever possible, it is suggested to incorporate any *a priori* knowledge of the search space into the initialization process to endow the genetic algorithm with a better starting point.

Step 2 Evaluation

Each chromosome is decoded and evaluated according to the given object function. The fitness value, f_i, reflects the degree of success chromosome c_i can achieve in its environment.

$$\begin{aligned} \vec{x}_i &= D(c_i) \\ f_i &= F(\vec{x}_i) \end{aligned} \tag{4}$$

Step 3 Selection

Chromosomes are stochastically picked to form the population for the next generation based on their fitness values. The selection is done by roulette wheel selection with replacement as the following:

$$Pr(c_i \text{ be selected}) = \frac{f_i^{SF}}{\sum_{j=1}^{N} f_j^{SF}} \tag{5}$$

The selection factor, *SF*, controls the discrimination between superior and inferior chromosomes by reshaping the landscape of the fitness function. As a result, better chromosomes will have more copies in the new population, mimicking the process of natural selection. In some applications, the best chromosome found is always retained in the next generation to ensure its genetic material remains in the gene pool.

Step 4 Crossover

Pairs of chromosomes in the newly generated population are subject to a crossover (or swap) operation with probability P_C, called Crossover Rate. The crossover operator generates new chromosomes by exchanging genetic material of pair of chromosomes across randomly selected sites, as depicted in Figure 3. Similar to the process of natural breeding, the newly generated chromosomes can be better or worse than their parents. They will be tested in the subsequent selection process, and only those which are an improvement will thrive.

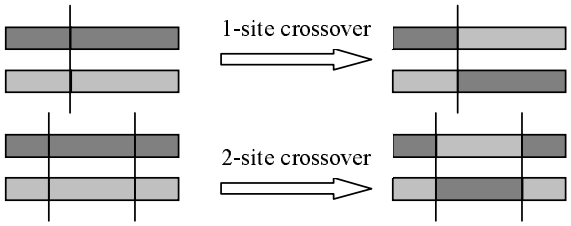


Figure 3. Crossover operation

Step 5 Mutation

After the crossover operation, each bit of all chromosomes are subjected to mutation with probability P_M, called the Mutation Rate. Mutation flips bit values and introduces new genetic material into the gene pool. This operation is essential to avoid the entire population converging to a single instance of a chromosome, since crossover becomes ineffective in such situations. In most applications, the mutation rate should be kept low and acts as a background operator to prevent genetic algorithms from random walking.

Step 6 Termination Checking

Genetic algorithms repeat Step 2 to Step 5 until a given termination criterion is met, such as pre-defined number of generations or quality improvement has failed to have progressed for a given number of generations. Once terminated, the algorithm reports the best chromosome it found.

Program 1 is an implementation of genetic algorithm. Note that, for the sake of program readability, variable of int type is used to store a single bit. More compact representation is possible with slightly tricky genetic operators.

The result of applying Program 1 to the object function F_1 is reported

in Figure 4. With a population of size 10, after 20 generations, the genetic algorithm was capable of locating a near optimal solution at $F_1(1.9853, 1.9810) = 4.2942$. Readers should be aware that, due to the stochastic nature of genetic algorithms, the same program may produce a different results on different machines.

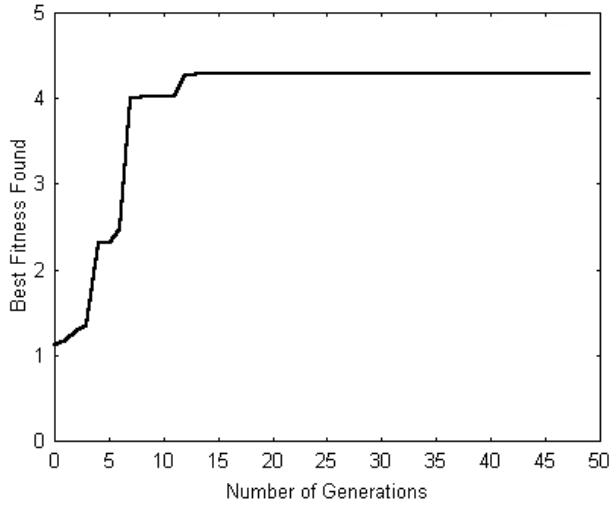


Figure 4. Progress of the GA program applied to test function F_1

Although the operation of genetic algorithms is quite simple, it does have some important characteristics providing robustness:

- They search from a population of points rather than a single point.
- The use the object function directly, not their derivative.
- They use probabilistic transition rules, not deterministic ones, to guide the search toward promising region.

In effect, genetic algorithms maintain a population of candidate solutions and conduct stochastic searches via information selection and

exchange. It is well recognized that, with genetic algorithms, near-optimal solutions can be obtained within justified computation cost. However, it is difficult for genetic algorithms to pin point the global optimum. In practice, a hybrid approach is recommended by incorporating gradient-based or local greedy optimization techniques. In such integration, genetic algorithms act as course-grain optimizers and gradient-based method as fine-grain ones.

The power of genetic algorithms originates from the chromosome coding and associated genetic operators. It is worth paying attention to these issues so that genetic algorithms can explore the search space more efficiently. The selection factor controls the discrimination between superior and inferior chromosomes. In some applications, more sophisticated reshaping of the fitness landscape may be required. Other selection schemes (Whitley 1993), such as rank-based selection, or tournament selection are possible alternatives for the controlling of discrimination.

Numerous variants with different application profiles have been developed following the standard genetic algorithm. Island-model genetic algorithms, or parallel genetic algorithms (Abramson and Abela 1991), attempt to maintain genetic diversity by splitting a population into several sub-populations, each of them evolves independently and occasionally exchanges information with each other. Multiple-objective genetic algorithms (Gao *et al.* 2000, Fonseca and Fleming 1993, Fonseca and Fleming 1998) attempt to locate all near-optimal solutions by careful controlling the number of copies of superior chromosomes such that the population will not be dominated by the single best chromosome (Sareni and Krahenbuhl 1998). Co-evolutionary systems (Handa *et al.* 2002, Bull 2001) have two or more independently evolved populations. The object function for each population is not static, but a dynamic function depends on the current states of other populations. This architecture vividly models interaction systems, such as prey and predator, virus and immune system.

3 Ant Systems

Inspired by the food-seeking behavior of real ants, Ant Systems, attributable to Dorigo *et al.* (Dorigo *et al.* 1996), has demonstrated itself to be an efficient, effective tool for combinatorial optimization problems. In nature, a real ant wandering in its surrounding environment will leave a biological trace, called pheromone, on its path. The intensity of left pheromone will bias the path-taking decision of subsequent ants. A shorter path will possess higher pheromone concentration and therefore encourage subsequent ants to follow it. As a result, an initially irregular path from nest to food will eventually contract to a shorter path. With appropriate abstraction and modification, this observation has led to a number of successful computational models for combinatorial optimization.

The operation of ant systems can be illustrated by the classical *Travelling Salesman Problem* (see Figure 5 for example). In the TSP problem, a travelling salesman problem is looking for a route which covers all cities with minimal total distance. Suppose there are n cities and m ants. The entire algorithm starts with initial pheromone intensity set to τ_0 on all edges. In every subsequent ant system cycle, or episode, each ant begins its trip from a randomly selected starting city and is required to visit every city exactly once (a Hamiltonian Circuit). The experience gained in this phase is then used to update the pheromone intensity on all edges.

The operation of ant systems is given below:

Step 1 Initialization
Initial pheromone intensities on all edges are set to τ_0.

Step 2 Walking phase
In this phase, each ant begins its trip from a randomly selected starting city and is required to visit every city exactly once. When an ant, the k-th ant for example, is located at city r and needs to

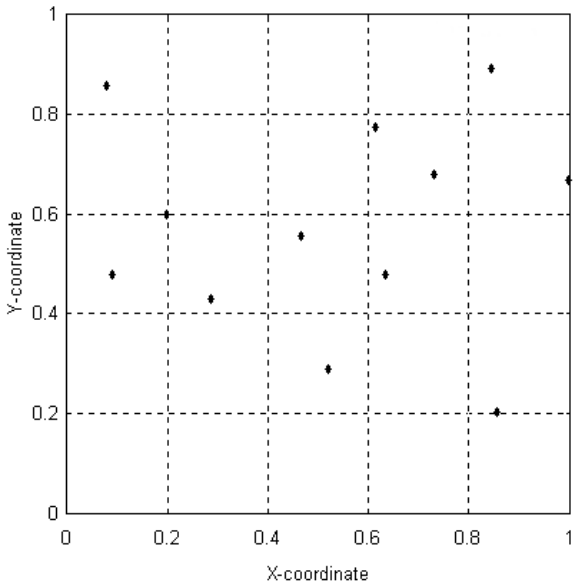


Figure 5. A traveling salesman problem with 12 cities

decide the next city s, the path-taking decision is made stochastically based on the following probability function:

$$P_k(r,s) = \begin{cases} \dfrac{[\tau(r,s)] \cdot [\eta(r,s)]^\beta}{\sum_{u \in J_k(r)} [\tau(r,u)] \cdot [\eta(r,u)]^\beta}, & \text{if } s \in J_k(r); \\ 0, & \text{otherwise.} \end{cases} \quad (6)$$

where $\tau(r,s)$ is the pheromone intensity on the edge between cities r and s; visibility $\eta(r,s) = \frac{1}{\delta(r,s)}$ is the reciprocal of the distance between cities r and s; $J_k(r)$ is the set of unvisited cities for the k-th ant. According to Equation 6, an ant will favour a nearer city or a path with higher pheromone intensity. β is parameter used to control the relative weighting of these two factors. During the circuit, the route made by each ant is recorded for pheromone updating in step 3. The best route found so far is also tracked.

Step 3 Updating phase

The experience accumulated in step 2 is then used to modify the pheromone intensity by the following updating rule:

$$\tau(r,s) \longleftarrow (1-\alpha)\cdot\tau(r,s) + \sum_{k=1}^{m} \Delta\tau_k(r,s) \qquad (7)$$

$$\Delta\tau_k(r,s) = \begin{cases} \frac{1}{L_k}, & \text{if } (r,s) \in \text{route made by ant } k; \\ 0, & \text{otherwise.} \end{cases}$$

where $0 < \alpha < 1$ is a parameter modelling the evaporation of pheromone; L_k is the length of the route made by the k-th ant; $\Delta\tau_k(r,s)$ is the pheromone trace contributed by the k-th ant to edge (r,s).

The updated pheromone intensities are then used to guide the path-taking decision in the next ant system cycle. It can be expected that, as the ant system cycle proceeds, the pheromone intensities on the edges will converge to values reflecting their potential for being components of the shortest route. The higher the intensity, the more chance of being a link in the shortest route, and *vice visa*.

Step 4 Termination Checking

Ant systems repeat Step 2 to Step 3 until certain termination criteria are met, such as a pre-defined number of episodes is performed or the algorithm has failed to make improvements for certain number of episodes. Once terminated, ant system reports the shortest route found.

Program 2 at the end of this chapter is an implementation of an ant system. The results of applying Program 2 to the test problem in Figure 5 are given in Figure 6 and 7. Figure 6 reports a found shortest route of length 3.308, which is the truly shortest route validated by exhaustive search. Figure 7 gives a snapshot of the pheromone intensities after 20 episodes. A higher intensity is represented by a

wider edge. Notice that intensity alone cannot be used as a criteria for judging whether a link is a constitute part of the shortest route or not, since the shortest route relies on the cooperation of other links.

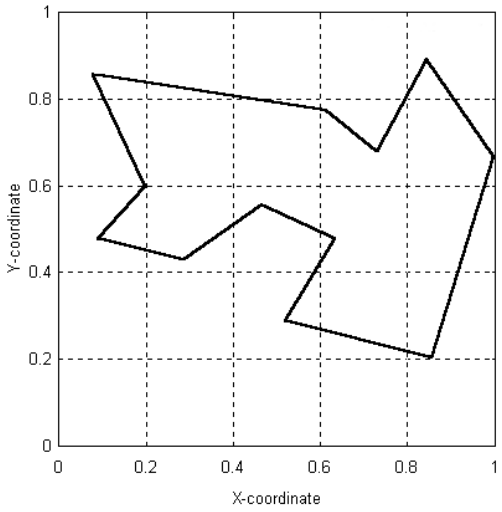


Figure 6. The shortest route found by the ant system

A close inspection on the ant system reveals that the heavy computation required may make it prohibitive in certain applications. Ant Colony Systems was introduced by Dorigo *et al.* (Dorigo and Gambardella 1997) to remedy this difficulty. Ant colony systems differ from the simpler ant system in the following ways:

- There is explicit control on exploration and exploitation. When an ant is located at city r and needs to decide the next city s, there are two modes for the path-taking decision, namely exploitation and biased exploration. Which mode to be used is governed by a random variable $0 < q < 1$,

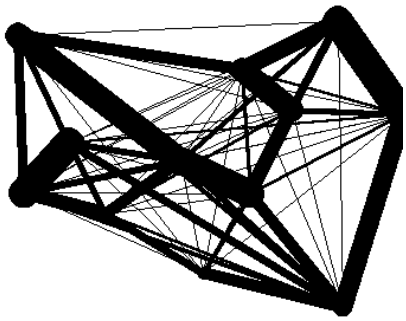


Figure 7. The snapshot of pheromone intensities after 20 episodes

Exploitation Mode:

$$s \quad = \quad \arg \max_{u \in J_k(r)} [\tau(r,u)] \cdot [\eta(r,u)]^{\beta}, \quad \text{if } q \leq q_0 \qquad (8)$$

Biased Exploration Mode:

$$P_k(r,s) \quad = \quad \frac{[\tau(r,s)] \cdot [\eta(r,s)]^{\beta}}{\sum_{u \in J_k(r)} [\tau(r,u)] \cdot [\eta(r,u)]^{\beta}}, \quad \text{if } q > q_0 \qquad (9)$$

- Local updating. A local updating rule is applied whenever a edge from city r to city s is taken:

$$\tau(r,s) \longleftarrow (1-\rho) \cdot \tau(r,s) + \rho \Delta\tau(r,s) \qquad (10)$$

where $\Delta\tau(r,s) = \tau_0 = (n \cdot L_{nn})^{-1}$, L_{nn} is a rough estimation of circuit length calculated using the nearest neighbor heuristic; $0 < \rho < 1$ is a parameter modeling the evaporation of pheromone.

- Count only the shortest route in global updating. As all ants complete their circuits, the shortest route found in the current episode is used in the global updating rule:

$$\tau(r,s) \longleftarrow (1-\alpha)\cdot\tau(r,s) + \alpha\Delta\tau(r,s) \tag{11}$$

$$\Delta\tau(r,s) = \begin{cases} (L_{gb})^{-1}, & \text{if } (r,s) \in \text{ global best route;} \\ 0, & \text{otherwise.} \end{cases}$$

  where L_{gb} is the length of shortest route.

In some respects, the ant system has implemented the idea of emergent computation – a global solution emerges as distributed agents performing local transactions, which is the working paradigm of real ants. The success of ant systems in combinatorial optimization makes it a promising tool for dealing with a large set of problems in the NP-complete class (Papadimitriou and Steiglitz 1982). In addition, the work of Wang and Wu (Wang and Wu 2001) has extended the applicability of ant systems further into continuous search space. Chu *et al.* (2003) have proposed a parallel ant colony system, in which groups of ant colonies explore the search space independently and exchange their experiences at certain time intervals.

4 Particle Swarm Optimization

Some social systems of natural species, such as flocks of birds and schools of fish, possess interesting collective behavior. In these systems, globally sophisticated behavior emerges from local, indirect communication amongst simple agents with only limited capabilities.

In an attempt to simulate this flocking behavior by computers, Kennedy and Eberthart (1995) realized that an optimization problem can be formulated as that of a flock of birds flying across an

area seeking a location with abundant food. This observation, together with some abstraction and modification techniques, led to the development of a novel optimization technique – particle swarm optimization.

Particle swarm optimization optimizes an object function by conducting a population-based search. The population consists of potential solutions, called particles, which are a metaphor of birds in flocks. These particles are randomly initialized and freely fly across the multi-dimensional search space. During flight, each particle updates its velocity and position based on the best experience of its own and the entire population. The updating policy will drive the particle swarm to move toward region with higher object value, and eventually all particles will gather around the point with highest object value. The detailed operation of particle swarm optimization is given below:

Step 1 Initialization

The velocity and position of all particles are randomly set to within pre-specified or legal range.

Step 2 Velocity Updating

At each iteration, the velocity of all particles are updated according to the following rule:

$$\vec{\nu}_i \longleftarrow \omega \cdot \vec{\nu}_i + c_1 \cdot R_1 \cdot (\vec{p}_{i,\text{best}} - \vec{p}_i) + c_2 \cdot R_2 \cdot (\vec{g}_{\text{best}} - \vec{p}_i) \quad (12)$$

where $\vec{p}_i$ and $\vec{\nu}_i$ are the position and velocity of particle i, respectively; $\vec{p}_{i,\text{best}}$ and $\vec{g}_{\text{best}}$ is the position with the 'best' object value found so far by particle i and the entire population, respectively; ω is a parameter controlling the dynamics of flying; R_1 and R_2 are random variables from the range $[0, 1]$; c_1 and c_2 are factors used to control the related weighting of corresponding terms.

The inclusion of random variables endows the particle swarm optimization with the ability of stochastic searching. The weight-

ing factors, c_1 and c_2, compromises the inevitable tradeoff between exploration and exploitation. After the updating, $\vec{\nu}_i$ should be checked and clamped to pre-specified range to avoid violent random walking.

Step 3 Position Updating

Assuming a unit time interval between successive iterations, the positions of all particles are updated according to the following rule:

$$\vec{p}_i \longleftarrow \vec{p}_i + \vec{\nu}_i \tag{13}$$

After updating, $\vec{p}_i$ should be checked and coerced to the legal range to ensure legal solutions.

Step 4 Memory Updating

Update $\vec{p}_{i,\text{best}}$ and $\vec{g}_{\text{best}}$ when condition is meet.

$$\begin{aligned} \vec{p}_{i,\text{best}} &\longleftarrow \vec{p}_i \quad \text{if } f(\vec{p}_i) > f(\vec{p}_{i,\text{best}}), \\ \vec{g}_{\text{best}} &\longleftarrow \vec{p}_i \quad \text{if } f(\vec{p}_i) > f(\vec{g}_{\text{best}}) \end{aligned} \tag{14}$$

where $f(\vec{x})$ is the object function subject to maximization.

Step 5 Termination Checking

The algorithm repeats Steps 2 to 4 until certain termination conditions are met, such as pre-defined number of iterations or a failure to make progress for certain number of iterations. Once terminated, the algorithm reports the $\vec{g}_{\text{best}}$ and $f(\vec{g}_{\text{best}})$ as its solution.

Program 3 at the end of this chapter is a straightforward implementation of the algorithm above. To experience the power of particle swarm optimization, Program 3 is applied to the following test function, as visualized in Figure 8.

$$F_2(x,y) = -x\sin\left(\sqrt{|\ x\ |}\right) - y\sin\left(\sqrt{|\ y\ |}\right),\ -500 < x, y < 500 \tag{15}$$

where global optimum is at $F_2(-420.97, -420.97) = 837.97$.

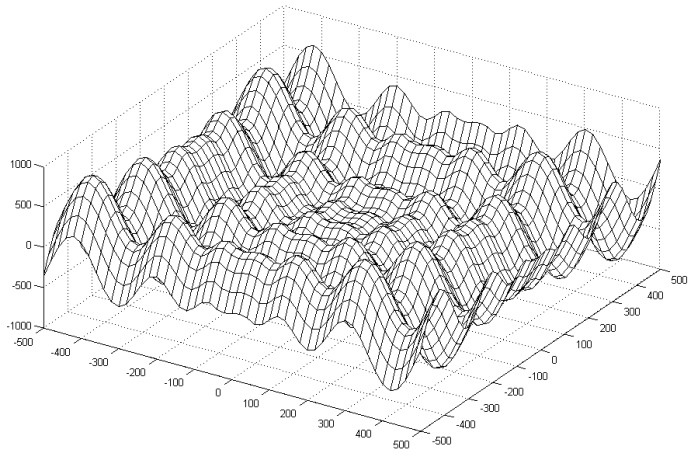


Figure 8. Object function F_2

In the tests above, both learning factors, c_1 and c_2, are set to a value of 2, and a variable inertia weight w is used according to the suggestion from Shi and Eberhart (1999). Figure 9 reports the progress of particle swarm optimization on the test function $F_2(x, y)$ for the first 300 iterations. At the end of 1000 iterations, $F_2(-420.97, -420.96) = 837.97$ is located, which is close to the global optimum.

It is worthwhile to look into the dynamics of particle swarm optimization. Figure 10 presents the distribution of particles at different iterations. There is a clear trend that particles start from their initial positions and fly toward the global optimum.

Numerous variants had been introduced since the first particle swarm

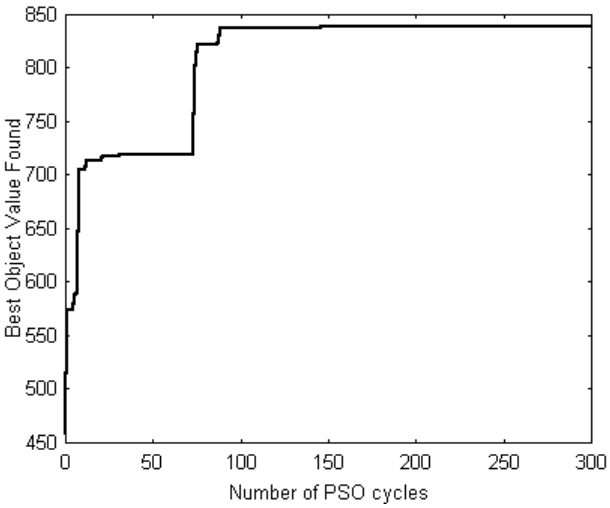
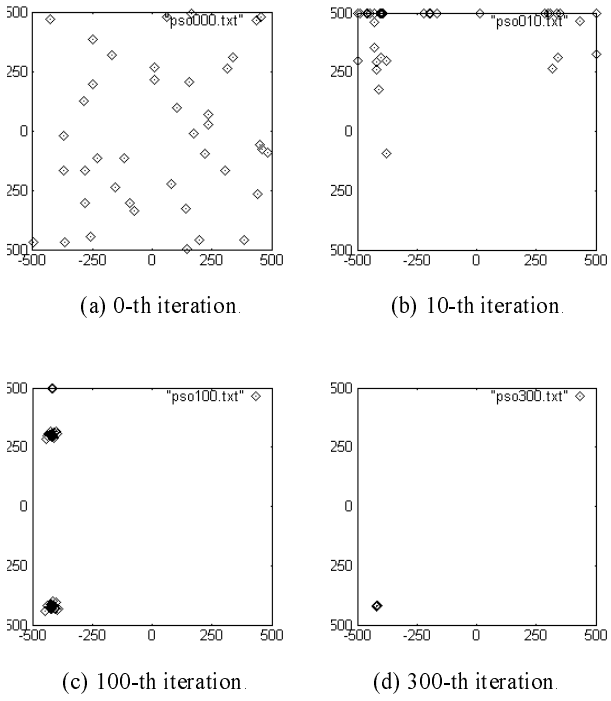


Figure 9. Progress of PSO on object function F_2

optimization. A discrete binary version of the particle swarm optimization algorithm was proposed by Kennedy and Eberhart (1997). Shi and Eberhart (2001) applied fuzzy theory to particle swarm optimization algorithm, and successfully incorporated the concept of co-evolution in solving min-max problems (Shi and Krohling 2002). (Chu *et al.* 2003) have proposed a parallel architecture with communication mechanisms for information exchange among independent particle groups, in which solution quality can be significantly improved.

5 Discussions and Conclusions

The nonstop evolution process has successfully driven natural species to develop effective solutions to a wide range of problems.



(a) 0-th iteration. (b) 10-th iteration.

(c) 100-th iteration. (d) 300-th iteration.

Figure 10. The distribution of particles at different iterations

Genetic algorithms, ant systems, and particle swarm optimization, all inspired by nature, have also proved themselves to be effective solutions to optimization problems. However, readers should remember that, despite of the robustness these approaches claim to be of, there is no panacea. As discussed in previous sections, there are control parameters involved in these meta-heuristics and an adequate setting of these parameters is a key point for success. In general, some kind of trial-and-error tuning is necessary for each particular instance of optimization problem. In addition, these meta-heuristics should not be considered in isolation. Prospective users should speculate on the possibility of hybrid approaches and the integration of gradient-based methods, which are promising directions deserving further study.

References

Abramson, D. and Abela, J. (1991), "A parallel genetic algorithm for solving the school timetabling problem," Technical report, Division of Information Technology, CSIRO.

Bull, L. (2001), "On coevolutionary genetic algorithms," *Soft Computing*, vol. 5, no. 3, pp. 201-207.

Chu, S. C. and Roddick, J. F. and Pan, J. S. (2003), "Parallel particle swarm optimization algorithm with communication strategies," personal communication.

Chu, S. C. and Roddick, J. F. and Pan, J. S. and Su, C. J. (2003), "Parallel ant colony systems," *14th International Symposium on Methodologies for Intelligent Systems*, LNCS, Springer-Verlag, (will appear in Japan).

Dorigo, M. and Maniezzo, V. and Colorni, A. (1996), "The ant system: optimization by a colony of cooperating agents," *IEEE Trans. on Systems, Man, and Cybernetics–Part B*, vol. 26, no. 2, pp. 29-41.

Dorigo, J. M. and Gambardella, L. M. (1997), "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Trans. on Evolutionary Computation*, vol. 26, no. 1, pp. 53–66.

Fonseca, C. M. and Fleming, P. J. (1993), "Multiobjective genetic algorithms," *IEE Colloquium on Genetic Algorithms for Control Systems Engineering*, number 1993/130, pp. 6/1-6/5.

Fonseca, C. M. and Fleming, P. J. (1998), "Multiobjective optimization and multiple constraint handling with evolutionary algorithms I: A unified formulation," *IEEE Trans. on Systems, Man and Cybernetics-Part A*, vol. 28, no. 1, pp. 26-37.

Gao, Y., Shi, L., and Yao, P. (2000), "Study on multi-objective genetic algorithm," *Proceedings of the Third World Congress on Intelligent Control and Automation*, pp. 646-650.

Goldberg, D. E. (1989), *Genetic Algorithm in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA.

Handa, H., Baba, M., Horiuchi, T., and Katai, O. (2002), "A novel hybrid framework of coevolutionary GA and machine learning," *International Journal of Computational Intelligence and Applications*, vol. 2, no. 1, pp. 33-52.

Holland, J. (1975), *Adaptation In Natural and Artificial Systems*, University of Michigan Press.

Kennedy, J. and Eberhart, R. (1995), "Particle swarm optimization," *IEEE International Conference on Neural Networks.*, pp. 1942-1948.

Papadimitriou C. H. and Steiglitz, K. (1982), *Combinatorial Optimization – Algorithms and Complexity*, Prentice Hall.

Sareni, B. and Krahenbuhl, L. (1998), "Fitness sharing and niching methods revisited," *IEEE Trans. on Evolutionary Computation*, vol. 2, no. 3, pp. 97-106.

Shi, Y. and Eberhart, R. C. (2001), "Fuzzy adaptive particle swarm optimization," *Proceedings of 2001 Congress on Evolutionary Computation (CEC'2001)*, pp. 101-106.

Shi, Y. and Krohling, R. A. (2002), "Co-evolutionary particle swarm optimization to solve min-max problems," *Proceedings of 2002 Congress on Evolutionary Computation (CEC'2002)*, vol. 2, pp. 1682-1687.

Wang, L. and Wu, Q. (2001), "Ant system algorithm for optimization in continuous space," *IEEE International Conference on Control Applications (CCA'2001)*, pp. 395–400.

Whitley, D. (1993), *A genetic algorithm tutorial*, Technical Report CS-93-103, Department of Computer Science, Colorado State University, Fort Collins, CO 8052.

Willis, M.-J., Hiden, H. G., Marenbach, P., McKay, B., and Montague, G.A. (1997), "Genetic programming: an introduction and survey of applications," *Proceedings of the Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'97)*, pp. 314-319.

Program 1. An implementation of genetic algorithm in C language.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MG 50    /* Maximal Number of Generations */
#define N  10    /* Population Size */
#define CL 32    /* Number of bits in each
                    chromosome */
#define SF 2.0   /* Selection Factor */
#define CR 0.5   /* Crossover Rate */
#define MR 0.05  /* Mutation Rate */

/* Macro for random number between 0 and 1 */
#define RAND ((float)rand()/(float)(RAND_MAX+1))

int   c[N][CL];      /* Population of Chromosomes */
float f[N];          /* Fitness Value of Chromosomes
*/
int   best_c[CL];    /* Best Chromosome */
float best_f;        /* Best Fitness Value */

/* Decode Chromosome */
void decode(int chromosome[CL],float *x,float *y)
{
int j;
/* Decode the lower 16 bits for variable x */
(*x)=0.0;
for(j=0;j<CL/2;j++)
    (*x)=(*x)*2.0+chromosome[j];
(*x)=(*x)/pow(2.0,16.0)*10.0-5.0;
/* Decode the upper 16 bits for variable y */
(*y)=0.0;
for(j=CL/2;j<CL;j++)
```

```
    (*y)=(*y)*2.0+chromosome[j];
(*y)=(*y)/pow(2.0,16.0)*10.0-5.0;
}

/* Object Function */
float object(float x,float y)
{
return(4.0/((x-2.0)*(x-2.0)+(y-2.0)*(y-2.0)+1.0)+3.
0/((x-2.0)*(x-2.0)+(y+2.0)*(y+2.0)+1.0)+2.0/((x+2.0
)*(x+2.0)+(y-2.0)*(y-2.0)+1.0));
}

void main(void)
{
int    i,k;         /* Index for Chromosome */
int    j;           /* Index for Generation */
int    gen;         /* Index for Generation */
float  x,y;         /* Independent Variables */
int    site;        /* Mutation Site */
float  tmpf;
int    tmpi;
int    tmpc[N][CL]; /* Temporary Population */
float  p[N];        /* Selection Probability */

/* Set random seed */
srand(4);

/* Initialize Population */
best_f=-1.0e99;
for(i=0;i<N;i++)
    {
    /* Randomly set each gene to '0' or '1' */
    for(j=0;j<CL;j++)
        if(RAND<0.5)
            c[i][j]=0;
        else
            c[i][j]=1;
```

```
    }

/* Repeat Genetic Algorithm cycle for MG times */
for(gen=0;gen<MG;gen++)
    {
    /* Evaluation */
    for(i=0;i<N;i++)
        {
        decode(c[i],&x,&y);
        f[i]=object(x,y);
        /* Update best solution */
        if(f[i]>best_f)
            {
            best_f=f[i];
            for(j=0;j<CL;j++)
                best_c[j]=c[i][j];
            }
        }
    /* Selection */
    /* Evaluate Selection Probability */
    tmpf=0.0;
    for(i=0;i<N;i++)
        {
        p[i]=pow(f[i],SF);
        tmpf=tmpf+p[i];
        }
    for(i=0;i<N;i++)
        p[i]=p[i]/tmpf;
    /* Retain the best Chromosome found so far */
    for(j=0;j<CL;j++)
        tmpc[0][j]=best_c[j];
    /* Roulette wheel selection with replacement */
    for(i=1;i<N;i++)
        {
        tmpf=RAND;
        for(k=0;tmpf>p[k];k++)
            tmpf=tmpf-p[k];
```

```
            /* Chromosome k is selected */
            for(j=0;j<CL;j++)
                tmpc[i][j]=c[k][j];
            }
        /* Copy temporary population to population */
        for(i=0;i<N;i++)
            for(j=0;j<CL;j++)
                c[i][j]=tmpc[i][j];
        /* 1-site Crossover */
        for(i=0;i<N;i=i+2)
            if(RAND<CR)
                {
                site=RAND*CL;
                for(j=0;j<site;j++)
                    {
                    tmpi=c[i][j];
                    c[i][j]=c[i+1][j];
                    c[i+1][j]=tmpi;
                    }
                }
        /* Mutation */
        for(i=0;i<N;i++)
            for(j=0;j<CL;j++)
                if(RAND<MR)
                      /* Flip j-th gene */
                      c[i][j]=1-c[i][j];
        /* Report Progress */
        printf("%f\n",best_f);
        }
    /* Report Solution */
    decode(best_c,&x,&y);
    printf("F(%f,%f)=%f\n",x,y,object(x,y));
    }
```

Program 2. An implementation of ant system in C language.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define Map     "map.txt"    /* file name of city ma
p */
#define NumberOfCity    12   /* number of cities */
#define NumberOfAnt     10   /* number of ants */
#define alpha           0.2  /* pheromone decay fact
or */
#define beta            2.0  /* tradeoff factor
                between pheromone and distance */
#define tau0            0.01 /* initial intensity of
                                pheromone */
#define EpisodeLimit    20   /* limit of episode */
#define Route  "route.txt"   /* file name for route
                                map */
/* RAND: Macro for random number between 0 and 1 */
#define RAND ((float)rand()/(float)(RAND_MAX+1))

typedef struct {
   float x;     /* x coordinate */
   float y;     /* y coordinate */
   } CityType;
typedef struct {
   int    route[NumberOfCity];
                  /* visiting sequence of cities */
   float length;               /* length of route */
   } RouteType;

CityType  city[NumberOfCity];        /* city array */
float    delta[NumberOfCity][NumberOfCity];
                   /* distance matrix */
float    eta[NumberOfCity][NumberOfCity];
```

```
                    /* weighted visibility matrix */
float    tau[NumberOfCity][NumberOfCity];
                    /* pheromone intensity matrix */
RouteType BestRoute;          /* shortest route */
RouteType ant[NumberOfAnt];   /* ant array */

float  p[NumberOfCity];        /* path-taking proba-
                                  bility array */
int   visited[NumberOfCity];  /* array for visiting
status */

float    delta_tau[NumberOfCity][NumberOfCity];
                          /* sum of change in tau */

void main(void)
{
FILE*   mapfpr;      /* file pointer for city map */
int     r,s;         /* indices for cities */
int     k;           /* index for ant */
int     episode;     /* index for ant system cycle */
int     step;        /* index for routing step */
float   tmp;         /* temporary variable */
FILE*   routefpr;    /* file pointer for route map */

/* Set random seed */
srand(1);

/* Read city map */
mapfpr=fopen(Map,"r");
for(r=0;r<NumberOfCity;r++)
       fscanf(mapfpr,"%f %f",&(city[r].x),&(city[r].
y));
fclose(mapfpr);

/* Evaluate distance matrix */
for(r=0;r<NumberOfCity;r++)
       for(s=0;s<NumberOfCity;s++)
```

```
            delta[r][s]=sqrt((city[r].x-city[s].x)
*(city[r].x-city[s].x)+(city[r].y-city[s].y)*(city[
r].y-city[s].y));

/* Evaluate weighted visibility matrix */
for(r=0;r<NumberOfCity;r++)
      for(s=0;s<NumberOfCity;s++)
            if(r!=s)
                  eta[r][s]=pow((1.0/delta[r][s])
,beta);
            else
                  eta[r][s]=0.0;

/* Initialize pheromone on edges */
for(r=0;r<NumberOfCity;r++)
      for(s=0;s<NumberOfCity;s++)
            tau[r][s]=tau0;

/* Initialize best route */
BestRoute.route[0]=0;
BestRoute.length=0.0;
for(r=1;r<NumberOfCity;r++)
      {
      BestRoute.route[r]=r;
      BestRoute.length+=delta[r-1][r];
      }
BestRoute.length+=delta[NumberOfCity-1][0];

/* Repeat ant system cycle for EpisodeLimit times */
for(episode=0;episode<EpisodeLimit;episode++)
      {
      /* Initialize ants' starting city */
      for(k=0;k<NumberOfAnt;k++)
            ant[k].route[0]=RAND*NumberOfCity;
```

```
/* Let all ants proceed for NumberOfCity-1 steps */
      for(step=1;step<NumberOfCity;step++)
            {
            for(k=0;k<NumberOfAnt;k++)
                  {
               /* Evaluate path-taking probability
               array for ant k at current time step
               */
                   r=ant[k].route[step-1];
                /* Clear visited list of ant k*/
                   for(s=0;s<NumberOfCity;s++)
                       visited[s]=0;
                /* Mark visited cities of ant k*/
                   for(s=0;s<step;s++)
                       visited[ant[k].route[s]]=1;
                   tmp=0.0;
                   for(s=0;s<NumberOfCity;s++)
                       if(visited[s]==1)
                              p[s]=0.0;
                       else
                              {
                              p[s]=tau[r][s]*eta[r]
                                    [s];
                              tmp+=p[s];
                              }
                   for(s=0;s<NumberOfCity;s++)
                          p[s]/=tmp;
                /* Probabilistically pick up next
                 edge by roulette wheel selection */
                   tmp=RAND;
                   for(s=0;tmp>p[s];s++)
                          tmp-=p[s];
                   ant[k].route[step]=s;
                   }
            }
```

```
/* Update pheromone intensity */
/* Reset matrix for sum of change in tau */
for(r=0;r<NumberOfCity;r++)
       for(s=0;s<NumberOfCity;s++)
              delta_tau[r][s]=0.0;
for(k=0;k<NumberOfAnt;k++)
       {
       /* Evaluate route length */
       ant[k].length=0.0;
       for(r=1;r<NumberOfCity;r++)
              ant[k].length+=delta[ant[k].
                                  route[r-1]]
                           [ant[k].route[r]];
       ant[k].length+=delta[ant[k].route
     [NumberOfCity-1]][ant[k].route[0]];

       /* Evaluate contributed delta_tau */
       for(r=1;r<NumberOfCity;r++)
          delta_tau[ant[k].route[r-1]][ant[k].
              route[r]]+=1.0/ant[k].length;
          delta_tau[ant[k].route[NumberOfCity
              -1]][ant[k].
              route[0]]+=1.0/ant[k].length;

       /* Update best route */
       if(ant[k].length<BestRoute.length)
              {
              BestRoute.length=ant[k].length;
              for(r=0;r<NumberOfCity;r++)
                     BestRoute.route[r]=ant[k]
                                  .route[r];
              }
       }
/* Update pheromone matrix */
for(r=0;r<NumberOfCity;r++)
       for(s=0;s<NumberOfCity;s++)
```

```
                    tau[r][s]=(1.0-alpha)*tau[r][s]
                                +delta_tau[r][s];
        printf("%d %f\n",episode,BestRoute.length);
        }

/* Write route map */
routefpr=fopen(Route,"w");
for(r=0;r<NumberOfCity;r++)
        fprintf(routefpr,"%f %f\n",city[BestRoute
           .route[r]].x,city[BestRoute.route[r]].y);
fclose(routefpr);
}
```

Sample data, map.txt, for Program 2.

```
0.997375 0.666321
0.731415 0.678162
0.078674 0.856720
0.854553 0.204895
0.199432 0.599670
0.613770 0.773651
0.287476 0.430420
0.518707 0.288818
0.465729 0.555573
0.843231 0.890839
0.091431 0.479309
0.634186 0.478363
```

Program 3. An implementation of particle swarm optimi zation in C language.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define IterationLimit  1000   /* Maximal Number of I
teration */
#define PopulationSize  40     /* Population Size:
                                  Number of Particles */
#define Dimension     2        /* Dimension of Search
 Space */
#define wU            0.9      /* Upper Bound of Iner
tia Weight */
#define wL            0.4      /* Lower Bound of Iner
tia Weight */
#define c1            2.0      /* Acceleration Factor
 1 */
#define c2            2.0      /* Acceleration Factor
 2 */
#define Vmax       1000.0      /* Maximal Velocity */
#define RAND          ((float)rand()/(float)(RAND_
                                          MAX+1))

typedef struct {
   float x[Dimension];         /* Position */
   float v[Dimension];         /* Velocity */
   float fitness;              /* Fitness */
   float best_x[Dimension];    /* Individual Best
                                  Solution */
   float best_fitness;         /* Individual Best
                                  Fitness */
   } ParticleType;
```

```
ParticleType    p[PopulationSize];      /* Particle
                                           Array */
float           gbest_x[Dimension];     /* Global
                                      Best Solution */
float           gbest_fitness;          /* Global
                                       Best Fitness */

/* Schwefel Function */
float Schwefel(float x[Dimension])
{
int i;
float tmp;
tmp=0.0;
for(i=0;i<Dimension;i++)
       tmp+=(-1.0)*x[i]*sin(sqrt(fabs(x[i])));
return(tmp);
}

void main(void)
{
int     i;      /* Index for Particle */
int     d;      /* Index for Dimension */
float   w;      /* Inertia Weight */
int     step;   /* Index for PSO cycle */

/* Set random seed */
srand(1);

/* Initialize particles */
gbest_fitness=-1.0e99;
for(i=0;i<PopulationSize;i++)
    {
    for(d=0;d<Dimension;d++)
        {
        p[i].x[d]=p[i].best_x[d]=RAND*1000.0-500.0;
        p[i].v[d]=RAND*1000.0-500.0;
        }
```

```
        p[i].fitness=p[i].best_fitness=Schwefel(p[i]
                    .x);
        /* Update gbest */
        if(p[i].best_fitness>gbest_fitness)
               {
               for(d=0;d<Dimension;d++)
                      gbest_x[d]=p[i].best_x[d];
               gbest_fitness=p[i].best_fitness;
               }
        }

/* Repeat PSO cycle for InterationLimit times */
for(step=0;step<IterationLimit;step++)
   {
   w=wU-(wU-wL)*((float)step/(float)IterationLimit);
        for(i=0;i<PopulationSize;i++)
              {
              for(d=0;d<Dimension;d++)
                  {
                  p[i].v[d]=w*p[i].v[d]+c1*RAND*(p[i]
.best_x[d]-p[i].x[d])+c2*RAND*(gbest_x[d]-p[i].x[d]
);
                  if(p[i].v[d]>Vmax)p[i].v[d]=Vmax;
                  if(p[i].v[d]<-Vmax)p[i].v[d]=-Vmax;
                      p[i].x[d]=p[i].x[d]+p[i].v[d];
                      if(p[i].x[d]>500.0)p[i].x[d]
                      =500.0;
                      if(p[i].x[d]<-500.0)p[i].x[d]
                      =-500.0;                        }
              p[i].fitness=Schwefel(p[i].x);
              /* Update pbest */
              if(p[i].fitness>p[i].best_fitness)
                     {
                     for(d=0;d<Dimension;d++)
                            p[i].best_x[d]=p[i].x[d];
                     p[i].best_fitness=p[i].fitness;
```

```
                /* Update gbest */
                if(p[i].best_fitness>gbest_fitness)
                      {
                      for(d=0;d<Dimension;d++)
                         gbest_x[d]=p[i].best_x[d];
                         gbest_fitness=p[i].best_
                                       fitness;
                       }
                }
              }
       printf("%f\n",gbest_fitness);
       }
}
```