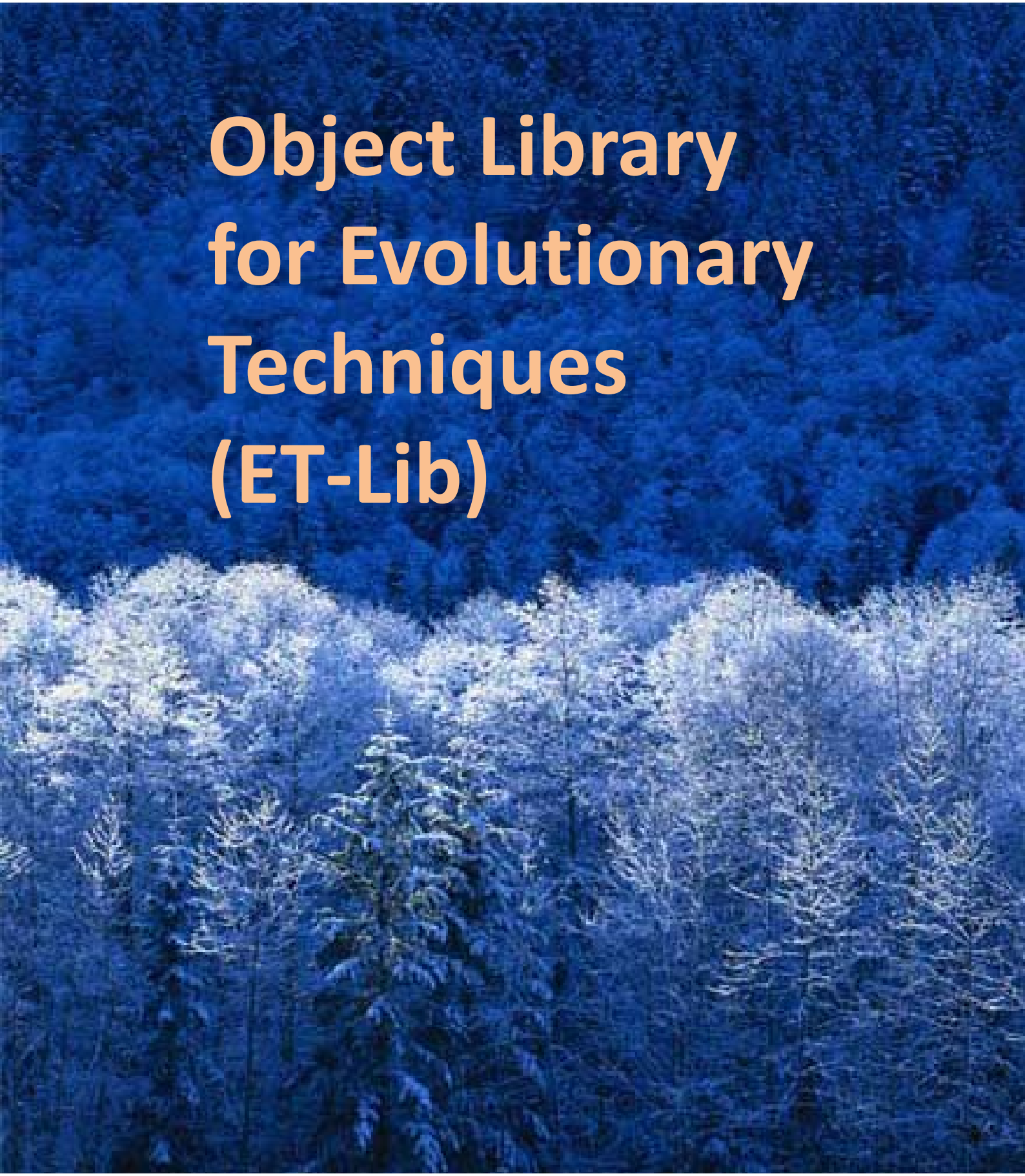

Object Library for Evolutionary Techniques (ET-Lib)

ASIAN INSTITUTE OF TECHNOLOGY

USER'S MANUAL

Object Library for Evolutionary Techniques (ET-Lib)

version 1.0

Su Nguyen
T. J. Ai
Voratas Kachitvichyanukul

Industrial & Manufacturing Engineering
School of Engineering & Technology
Asian Institute of Technology
THAILAND

April 2010

ACKNOWLEDGEMENT

The researchers working in the projects are supported by: Royal Thai Government Scholarship Program, the RTG-AIT Joint Research Funding 2008.

Key Contributors

Voratas Kachitvichyanukul
T. J. Ai
Su Nguyen

Content of Public Release 1.0

User's Manual, Version 1.0
Dynamic Link Library
Job Shop Scheduling Example
TSP Example,
Multi-objective PSO:
Portfolio Optimization
Beam Design Optimization

PREFACE

The first version of the library of Evolutionary Techniques (ET-Lib) was developed in 2008 at the Asian Institute of Technology (AIT), Thailand. The purpose of this library is to provide the researchers and students who are working on various optimization problems with a general and effective tool based on various evolutionary techniques. The first release contains mainly the Particle Swarm Optimization algorithm with multiple social learning terms (GLNPSO).

Currently, GLNPSO algorithm is completely written in C# as an object-oriented library. The library includes all the necessary classes and routines which can be used to implement the PSO algorithm. Users with little programming knowledge can still use classes provided in this ET-library to solve basic problems. For more complicated problems, it is recommended that the users are familiar with C# programming language at elementary level.

This manual is organized into 4 chapters. The first chapter will provide users who are new to the PSO concept the first introduction to this algorithm. Chapter 2 is used to explain the structure of the GLNPSO algorithm and a basic example are given to explain how to solve a simple problem with GLNPSO algorithm. In chapter 3, some practical applications of GLNPSO are presented with the introduction to such additional features as re-initialization, and multi-stage PSO. Finally, Chapter 4 discusses an extension of GLNPSO algorithm to deal with multi-objective optimization problems.

CHAPTER 1

INTRODUCTION TO PARTICLE SWARM OPTIMIZATION

1.1. Overview

Particle Swarm Optimization (PSO) is a population based random search method that imitates the physical movements of the individuals in the swarm as a searching mechanism. The first PSO algorithm was proposed by Kennedy and Eberhart in 1995. The key concept of PSO is to learn from the cognitive knowledge of each particle and the social knowledge of the swarm to guide particles to better position.

In the PSO algorithm, a solution of a specific problem is represented by an n-dimensional position of a particle. A swarm of fixed number of particles is generated and each particle is initialized with a random position in a multidimensional search space. Each particle flies through the multidimensional search space with a velocity. In each step of the iteration the velocity of each particle is adjusted based on three components:

- current velocity of the particle which represents the inertia term or momentum of the particle
- the position corresponds to the best solution achieved so far by the particle normally referred as personal best
- the position corresponds to the best solution achieved so far by all the particles, i.e., the global best

Once the velocity of each particle is updated, the particles are then moved to the new positions. The cycle repeats until the stopping criterion is met. The specific expressions used in the original particle swarm optimization algorithm will be discussed in the next section.

1.2. The Basic Form of PSO

The notations used to describe the algorithms are given here and followed by a summary description of the original PSO algorithm.

Notations:

τ	:	Iteration index; $\tau = 1 \ldots T$
l	:	Particle index, $l = 1 \ldots L$
h	:	Dimension index, $h = 1 \ldots H$
u	:	Uniform random number in the interval $[0,1]$
$w(\tau)$	:	Inertia weight in the τ^{th} iteration
$\omega_{lh}(\tau)$	:	Velocity of the l^{th} particle at the h^{th} dimension in the τ^{th} iteration
$\theta_{lh}(\tau)$	:	Position of the l^{th} particle at the h^{th} dimension in the τ^{th} iteration
ψ_{lh}	:	Personal best position (pbest) of the l^{th} particle at the h^{th} dimension
ψ_{gh}	:	Global best position (gbest) at the h^{th} dimension
c_p	:	Personal best position acceleration constant
c_g	:	Global best position acceleration constant

$\theta^{\max}$: Maximum position value
$\theta^{\min}$: Minimum position value
Θ_l : Vector position of the l^{th} particle, $\left[\theta_{l1} \quad \theta_{l2} \quad \cdots \quad \theta_{lH}\right]$
Ω_l : Vector velocity of the l^{th} particle, $\left[\omega_{l1} \quad \omega_{l2} \quad \cdots \quad \omega_{lH}\right]$
Ψ_l : Vector personal best position of the l^{th} particle, $\left[\psi_{l1} \quad \psi_{l2} \quad \cdots \quad \psi_{lH}\right]$
Ψ_g : Vector global best position, $\left[\psi_{g1} \quad \psi_{g2} \quad \cdots \quad \psi_{gH}\right]$
R_l The l^{th} set of solution
$Z(\Theta_l)$: Fitness value of Θ_l

Algorithm PSO

1. Initialize L particles as a swarm:
   Set iteration $\tau = 1$. Generate the l^{th} particle with random position $\Theta_l(\tau)$ in the range $\left[\theta^{\min}, \theta^{\max}\right]$, velocity $\Omega_l(\tau) = 0$ and personal best $\Psi_l = \Theta_l$ for $l = 1 \ldots L$.
2. Decode particles into solutions:
   For $l = 1 \ldots L$, decode $\Theta_l(\tau)$ to a solution R_l. (This step is only needed if the particles are not directly representing the solutions).
3. Evaluate the particles:
   For $l = 1 \ldots L$, compute the performance measurement of R_l, and set this as the fitness value of $\Theta_l(\tau)$, represented by $Z(\Theta_l)$.
4. Update pbest:
   For $l = 1 \ldots L$, update $\Psi_l = \Theta_l$, if $Z(\Theta_l) < Z(\Psi_l)$.
5. Update gbest:
   For $l = 1 \ldots L$, update $\Psi_g = \Psi_l$, if $Z(\Psi_l) < Z(\Psi_g)$.
6. Update the velocity and the position of each l^{th} particle:

$$w(\tau) = w(T) + \frac{\tau - T}{1 - T}\left[w(1) - w(T)\right] \tag{1.1}$$

$$\omega_{lh}(\tau+1) = w(\tau)\omega_{lh}(\tau) + c_p u\left(\psi_{lh} - \theta_{lh}(\tau)\right) + c_g u\left(\psi_{gh} - \theta_{lh}(\tau)\right) \tag{1.2}$$

$$\theta_{lh}(\tau+1) = \theta_{lh}(\tau) + \omega_{lh}(\tau+1) \tag{1.3}$$

If $\theta_{lh}(\tau+1) > \theta^{\max}$, then

$$\theta_{lh}(\tau+1) = \theta^{\max} \tag{1.4}$$

$$\omega_{lh}(\tau+1) = 0 \tag{1.5}$$

If $\theta_{lh}(\tau+1) < \theta^{\min}$, then

$$\theta_{lh}(\tau+1) = \theta^{\min} \tag{1.6}$$

$$\omega_{lh}(\tau+1)=0 \qquad (1.7)$$

7. If the stopping criterion is met, i.e., $\tau = T$, stop. Otherwise, $\tau = \tau + 1$ and return to step 2.

The basic version of PSO algorithm described above contains the inertia term with position boundary and linear decreasing weight introduced by Shi and Eberhart (1998) to explore the solution space in the initial phase and following the cognitive and social term to exploit the personal best and global best in the final phase. In addition, this algorithm is applicable for minimization problem.

1.3. Key parameters of PSO

This section discusses possible qualifications and effects of each parameter on the performance of PSO. The parameters analyzed in this section consist of the population size (L), two acceleration constants (c_p and c_g), and the inertia weight (w). The discussion is presented below.

Population size (L)

This parameter represents the number of particles in the system. It is one important parameter of PSO, because it affects the fitness value and computation time. Furthermore, increasing size of population always increases computation time, but might not improve the fitness value. Generally speaking, too small a population size can lead to poor convergence while too large a population size can yield good convergence at the expense of long running time.

Acceleration constants (c_p and c_g)

The constants c_p and c_g are the acceleration constants of the personal best position and the global best position, respectively. Each acceleration constant controls the maximum distance that a particle is allowed to move from the current position to each best position. The new velocity can be viewed as a vector which combines the current velocity, and the vectors of the best positions. Each best position's vector consists of the direction which is pointed from the particle's current position to the best position, and the magnitude of the movement can be between 0 to the acceleration constant of the best position times the distance between the best position and the current position.

Inertia weight (w)

The new velocity is produced from the combination of vectors. One of these vectors is the current velocity. Inertia weight is a weight to control the magnitude of the current velocity on updating the new velocity. For $w = c$, it means that this vector has the same direction of the current velocity, and the magnitude which equals to c times the current velocity's magnitude. This weight is one of the parameters to control the search behavior of the swarm.

Velocity boundary (V^{max}) and Position boundary (θ^{max})

Some PSO algorithms are implemented with bound on velocity. For each dimension, the magnitude of a velocity cannot be greater than V^{max}. This parameter is one of parameters to control the search behavior of the swarm. The smaller value of this parameter makes the particles in the population less aggressive in the search.

In the PSO particle movement mechanism, it is also common to limit the search space of particle location, i.e. the position value of particle dimension is bounded in the interval $\left[\theta^{\min},\theta^{\max}\right]$. The use of position boundary θ^{max} is to force each particle to move within the feasible region to avoid solution divergence. Hence, the position value of certain particle dimension is being set at the minimum or maximum value whenever it moves beyond the boundary. In addition, the velocity of the corresponding dimension is reset to zero to avoid further movement beyond the boundary.

More detailed discussions of PSO behaviors in literatures include Ozcan and Mohan (1999), Carlisle and Dozier (2000, 2001), Beielstein, Parsopoulos, and Vrahatis (2002).

1.4. GLNPSO

Pongchairerks and Kachitvichyanukul (2005, 2009) proposed PSO with multiple social structures that were built by combining previously published structures. There are two additional social structures which are local best (lbest) and near neighbor best (nbest); this structure was presented in Veeramachaneni et al. (2003). Local best receives the best fitness value from sub group; each particle can update the velocity based on the best performance of neighbors in the population that is related on indices of particles. Near neighbor best obtains the maximum Fitness Distance Ratio (FDR) among all other particles.

The GLNPSO algorithm was described below following the notation that was added from previous algorithm.

Notation

τ	:	Iteration index; $\tau = 1 \ldots T$
l	:	Particle index, $l = 1 \ldots L$
h	:	Dimension index, $h = 1 \ldots H$
u	:	Uniform random number in the interval $[0,1]$
$w(\tau)$	:	Inertia weight in the τ^{th} iteration
$\omega_{lh}(\tau)$	:	Velocity of the l^{th} particle at the h^{th} dimension in the τ^{th} iteration
$\theta_{lh}(\tau)$	:	Position of the l^{th} particle at the h^{th} dimension in the τ^{th} iteration
ψ_{lh}	:	Personal best position (pbest) of the l^{th} particle at the h^{th} dimension
ψ_{gh}	:	Global best position (gbest) at the h^{th} dimension
ψ_{lh}^{L}	:	Local best position (lbest) of the l^{th} particle at the h^{th} dimension
ψ_{lh}^{N}	:	Near neighbor best position (nbest) of the l^{th} particle at the h^{th}
c_p	:	Personal best position acceleration constant
c_g	:	Global best position acceleration constant
c_l	:	Local best position acceleration constant

c_n : Near neighbor best position acceleration constant
$\theta^{\max}$: Maximum position value
$\theta^{\min}$: Minimum position value
Θ_l : Vector position of the l^{th} particle, $\left[\theta_{l1} \quad \theta_{l2} \quad \cdots \quad \theta_{lH}\right]$
Ω_l : Vector velocity of the l^{th} particle, $\left[\omega_{l1} \quad \omega_{l2} \quad \cdots \quad \omega_{lH}\right]$
Ψ_l : Vector personal best position of the l^{th} particle, $\left[\psi_{l1} \quad \psi_{l2} \quad \cdots \quad \psi_{lH}\right]$
Ψ_g : Vector global best position, $\left[\psi_{g1} \quad \psi_{g2} \quad \cdots \quad \psi_{gH}\right]$
Ψ_l^L : Vector local best position of the l^{th} particle, $\left[\psi_{l1}^L \quad \psi_{l2}^L \quad \cdots \quad \psi_{lD}^L\right]$
R_l The l^{th} set of solution
$Z(\Theta_l)$: Fitness value of Θ_l
FDR : Fitness-distance-ratio

Algorithm GLNPSO

1. Initialize L particles as a swarm:
   Set iteration $\tau = 1$. Generate the l^{th} particle with random position $\Theta_l(\tau)$ in the range $\left[\theta^{\min}, \theta^{\max}\right]$, velocity $\Omega_l = 0$ and personal best $\Psi_l = \Theta_l$ for $l = 1 \ldots L$.
2. Decode particles into solutions:
   For $l = 1 \ldots L$, decode $\Theta_l(\tau)$ to a solution R_l. (This step is only needed if the particles are not directly representing the solutions).
3. Evaluate the particles:
   For $l = 1 \ldots L$, compute the performance measurement of R_l, and set this as the fitness value of Θ_l, represented by $Z(\Theta_l)$.
4. Update pbest:
   For $l = 1 \ldots L$, update $\Psi_l = \Theta_l$, if $Z(\Theta_l) < Z(\Psi_l)$.
5. Update gbest:
   For $l = 1 \ldots L$, update $\Psi_g = \Psi_l$, if $Z(\Psi_l) < Z(\Psi_g)$.
6. Update lbest:
   For $l = 1 \ldots L$, among all pbest from K neighbors of the l^{th} particle, set the personal best which obtains the least fitness value to be Ψ_l^L.
7. Generate nbest:
   For $l = 1 \ldots L$, and $h = 1 \ldots H$, set $\psi_{lh}^N = \psi_{oh}$ that maximizing fitness-distance-ratio (FDR) for $o = 1 \ldots L$. Where FDR is defined as
   $$FDR = \frac{Z(\Theta_l) - Z(\Psi_o)}{\left|\theta_{lh} - \psi_{oh}\right|} \text{ which } l \neq o \qquad (1.8)$$
8. Update the velocity and the position of each l^{th} particle:
   $$w(\tau) = w(T) + \frac{\tau - T}{1 - T}\left[w(1) - w(T)\right] \qquad (1.9)$$

$$\omega_{lh}(\tau+1) = w(\tau)\omega_{lh}(\tau) + c_p u\left(\psi_{lh} - \theta_{lh}(\tau)\right) + c_g u\left(\psi_{gh} - \theta_{lh}(\tau)\right) + c_l u\left(\psi_{lh}^{L} - \theta_{lh}(\tau)\right) + c_n u\left(\psi_{lh}^{N} - \theta_{lh}(\tau)\right) \quad (1.10)$$

$$\theta_{lh}(\tau+1) = \theta_{lh}(\tau) + \omega_{lh}(\tau+1) \quad (1.11)$$

If $\theta_{lh}(\tau+1) > \theta^{\max}$, then

$$\theta_{lh}(\tau+1) = \theta^{\max} \quad (1.12)$$

$$\omega_{lh}(\tau+1) = 0 \quad (1.13)$$

If $\theta_{lh}(\tau+1) < \theta^{\min}$, then

$$\theta_{lh}(\tau+1) = \theta^{\min} \quad (1.14)$$

$$\omega_{lh}(\tau+1) = 0 \quad (1.15)$$

9. If the stopping criterion is met, i.e. $\tau = T$, stop. Otherwise, $\tau = \tau + 1$ and return to step 2.

GLNPSO has been successfully applied to solve many NP-hard combinatorial problems. For examples, job shop scheduling problems, vehicle routing problems, multicommodity distribution network design problems, continuous (no-wait) flow shop problems, multi-mode resource constrained project scheduling problems, etc.

CHAPTER 2

STUCTURE OF GLNPSO LIBRARY

Before discussing each component in PSO library, we will demonstrate how GLNPSO work by a simple example. The source code of the example can be found in "\GLNPSO basic\Basic Models\PSO basic". The user can run this example with Microsoft Visual C# 2005 or the free Microsoft Visual C # 2008 Express Editions which is free to download at http://www.microsoft.com/express/download/.

2.1. First example

In this example, our objective is to minimize an objective function $f(\vec{x}) = \sum_{i=1}^{n}[0.01{x_i}^2 + 2 * \sin(x_i)\,] \, where\ \vec{x} = \{x_1, \dots, x_n\}, x_i \in [-100,100]\ with\ \forall i.$ The graph of this function with n=1 is shown in Figure. 2.1. This is an extensive version of sphere function which includes some noise to make it more interesting. Here, GLNPSO library is applied to find the optimal solution $\overrightarrow{x^*}$ to minimize$f(\vec{x})$. For the ease of interpretation of the results and the dynamic of PSO algorithm, we start by solving the problem with n=1.

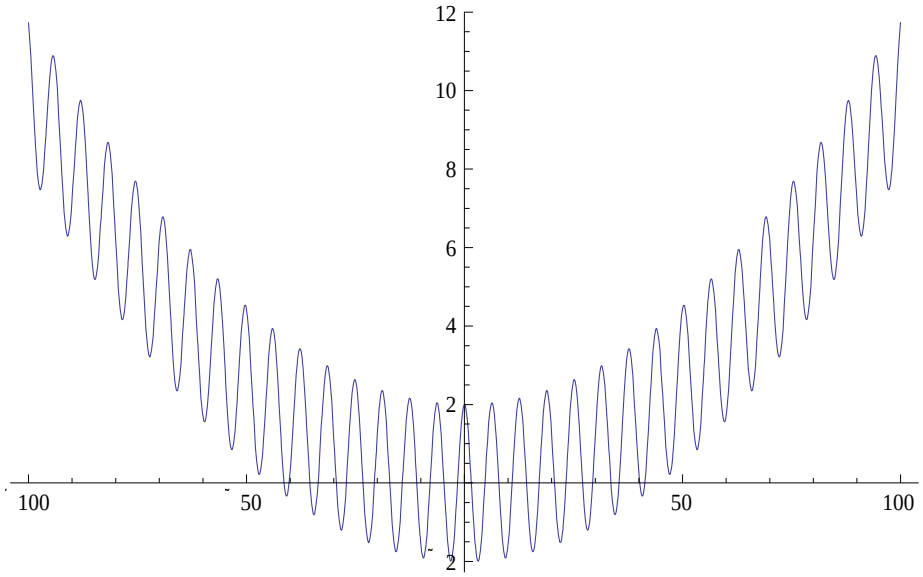

Figure 2.1: Function with multiple local minimum

For this simple problem, only problem formulation needs to be defined and this part is written in GLNPSO.cs. The implementation of GLNPSO on this problem is presented in Figure 2.2. In order to create a new class of PSO to solve a specific problem, three important questions needs to be clarified:

- What is the dimension of a particle?
- How to evaluate the fitness of a particle?
- How can the swarm be initialized?

In case that n=1, the position of a particle is defined as a real number x which ranges from -100 to 100 and the particle's dimension is 1. The objective function $f(\vec{x}) = f(x)$ is

used to measure the fitness of each particle (GLNPSO is designed to minimize the objective function, in case of maximization we just simply change the sign of the objective function to convert it to minimization problem). A particle is considered to be located at better position if its position results in a smaller objective value (in figure 2.2. the objective evaluation method is defined so that it can also handle the more generalized problem where n>1). The initial swarm is created by randomly generating the position of each particle in the swarm, which means that each position will follow the Uniform Distribution with the lower bound of -100 and upper bound of 100.

```
Class spPSO : PSO
{     // this part is the problem specific code
      // Minimize f(x) = 0.001x^2 + 2*Sin(x), -100<=x<=100
public spPSO(int nPar, int nIter, int nNB, double dwmax, double dwmin,
double dcp, double dcg, double dcl, double dcn):
base(nIter, nNB, dwmax, dwmin, dcp, dcg, dcl, dcn)
{
      base.SetDimension(nPar, 1);
}
//Define objective function
public override double Objective(Particle P)
{
      double obj = 0;
      for (int i = 0; i < P.Dimension; i++)
          obj += 0.001 * Math.Pow(P.Position[i], 2) + 2 *
                         Math.Sin(P.Position[i]);
      return obj;
}
//Initialize a swarm
public override void InitSwarm()
{
      for(int i=0; i<sSwarm.Member; i++)
      {
        for (int j = 0; j < sSwarm.pParticle[i].Dimension; j++)
        {
          sSwarm.pParticle[i].Position[j] = -100+200*rand.NextDouble();
          sSwarm.pParticle[i].Velocity[j] = 0;
          sSwarm.pParticle[i].BestP[j] = sSwarm.pParticle[i].Position[j];
          sSwarm.pParticle[i].PosMin[j] = -100;
          sSwarm.pParticle[i].PosMax[j] =  100;
        }
      sSwarm.pParticle[i].ObjectiveP = 1.7E308;
      }
      sSwarm.posBest = 0;
}
}
```

Dimension

Calculate Objective Value

Randomize initial positions and velocity and set boundary of position for each particle

Figure 2.2: Define a new PSO class for single variable example

Figure 2.3 shows the main class in which we define the PSO parameters and run the PSO algorithm with these parameters. In this experiment, only the global best and personal best is used to guide the swarm like the traditional PSO algorithm. The acceleration constants for local best and neighbor best are set to 0, and therefore the position of the local best and neighbor best do not influence the movement of particles in the swarm. The search space is explored by a swarm of size 10 in 200 iterations and three replications are performed. The final solutions and some statistics are reported in "MyPSO.xls" at the same folder of the execution file (\GLNPSO basic\PSO basic\PSO basic\bin\Debug).

```
class MainClass
{
public static void Main(string[] args)
{
int noPar = 10;
int noIter = 200;
int noNB = 5;
double wMax = 0.9;
double wMin = 0.4;
double cP = 2;
double cG = 2;
double cL = 0;
double cN = 0;
string oFile = "MyPSO.xls";
int noRep = 3
// starting time and finish time using DateTime datatype
DateTime start, finish;
// elapsed time using TimeSpan datatype
TimeSpan elapsed;

// opening output file
      TextWriter tw = new StreamWriter(oFile);
      tw.WriteLine("{0} Number of Particle  ", noPar);
      tw.WriteLine("{0} Number of Iteration ", noIter);
      tw.WriteLine("{0} Number of Neighbor  ", noNB);
      tw.WriteLine("{0} Parameter wmax      ", wMax);
      tw.WriteLine("{0} Parameter wmin      ", wMin);
      tw.WriteLine("{0} Parameter cp        ", cP);
      tw.WriteLine("{0} Parameter cg        ", cG);
      tw.WriteLine("{0} Parameter cl        ", cL);
      tw.WriteLine("{0} Parameter cn        ", cN);
      tw.WriteLine("{0} Output File Name    ", oFile);
      tw.WriteLine("");
for(int i=0; i<noRep; i++)
{
  Console.WriteLine("Replication {0}", i+1);
  tw.WriteLine("Replication {0}", i+1);
  // get the starting time from CPU clock
  start = DateTime.Now;

  // main program ...
  PSO myPSO = new spPSO(noPar, noIter, noNB, wMax, wMin, cP, cG, cL, cN);
  myPSO.Run(tw, true);
  myPSO.DisplayResult(tw);
  // get the finishing time from CPU clock
  finish = DateTime.Now;
  elapsed = finish - start;
  // display the elapsed time in hh:mm:ss.milli
  tw.WriteLine("{0} is the computational time", elapsed.Duration());
                        tw.WriteLine("");
}
tw.Close();
}
}
```

Define PSO parameters

Create new PSO object and pass the PSO parameters

Figure 2.3: Main class for single variable problem

In three replications, PSO needs less than 100 iterations to find the optimal solution. The average fitness and best fitness of a replication can be found in the output file and is presented in Figure 2.4 to show how fast PSO can converge to the optimal solution.

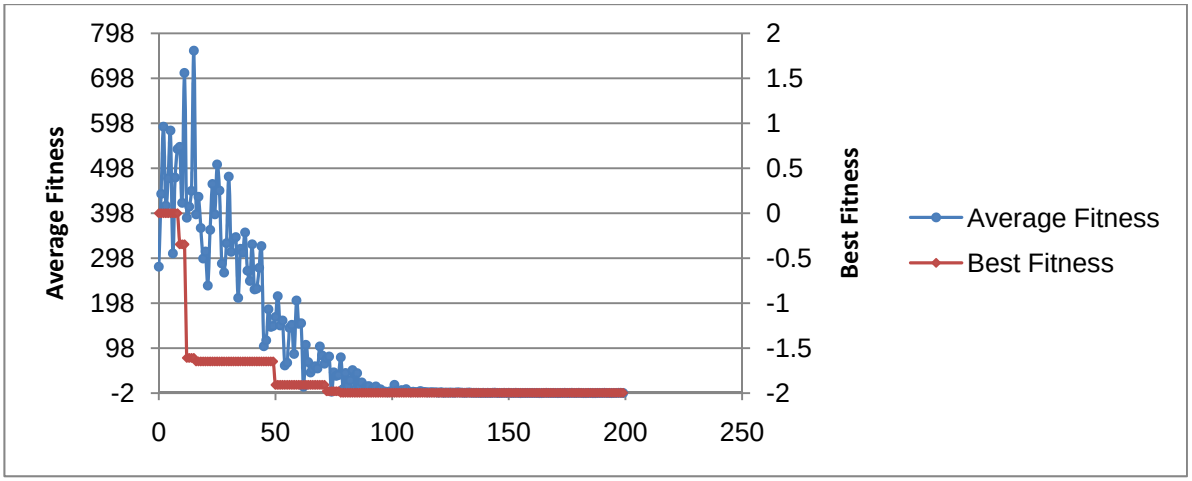

Figure 2.4: PSO performance in single variable problem

To help the reader have a better understanding of the dynamics of PSO algorithm, an animated version of this simple example is created (\GLNPSO basic\Basic Models\PSO_Visual). The screen shot of this application is shown in Figure 2.5. The user can choose the PSO parameters directly from the interface as well as select the function to be optimized. The upper left chart is to plot the function and the final solution found by PSO. The lower left chart shows the average of objective values for all the particles in the swarm at each iteration to check the convergence of the algorithm. In this application, users can perform a simple animation in x-y axis to observe the movement of the swarm during the searching process. The red circle points indicate current positions of particles. The personal best position of a particle is represented by an orange diamond point and finally, the green triangle point is the global best position found by the swarm. There are two options for animation so that the user can either choose to let the program automatically simulate all steps in PSO algorithm or run step by step (forward or backward) to observe the movement behavior carefully. In the step animation mode, the line connecting the position of a particle and its personal best position as well as the global best position is drawn to illustrate the direction for the movement. The user can exploit this feature to test the sensitivity of PSO parameters on the movement of the swarm. The 3D version of this application is also available at "\GLNPSO basic\Basic Models\PSO_Visual - 3D" as shown in Figure 2.6.

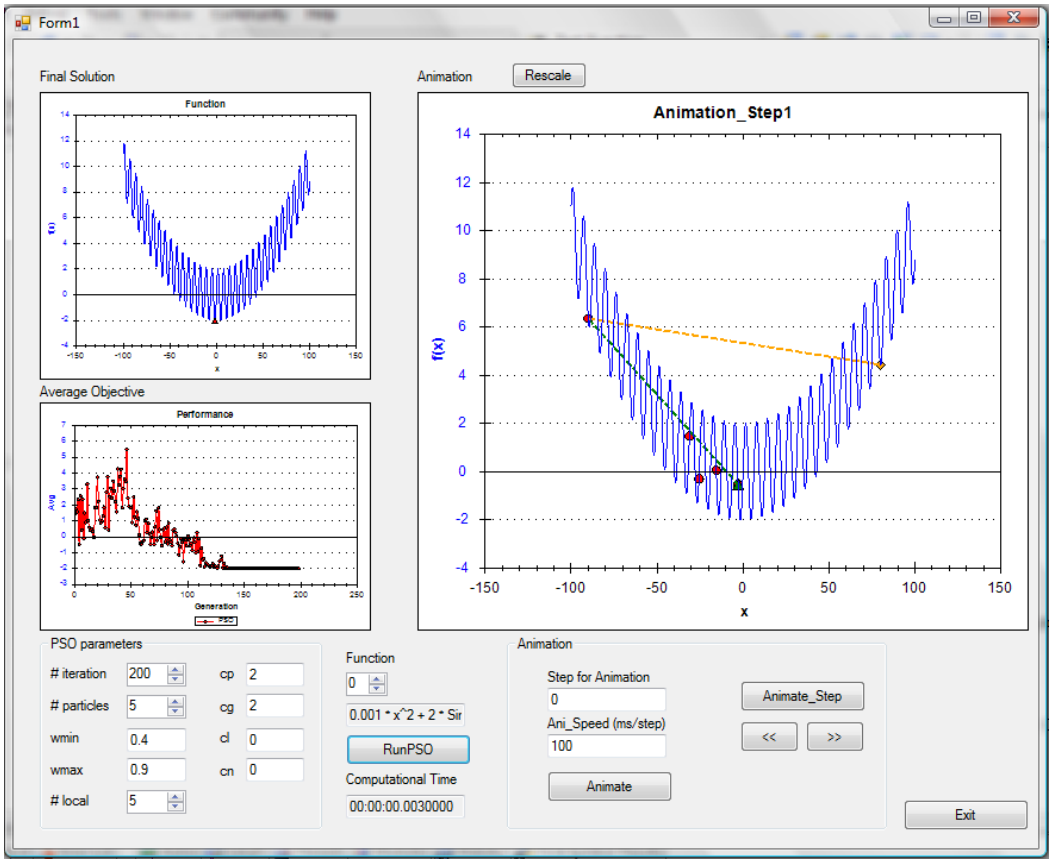

Figure 2.5: 2D Visual presentation of GLNPSO algorithm

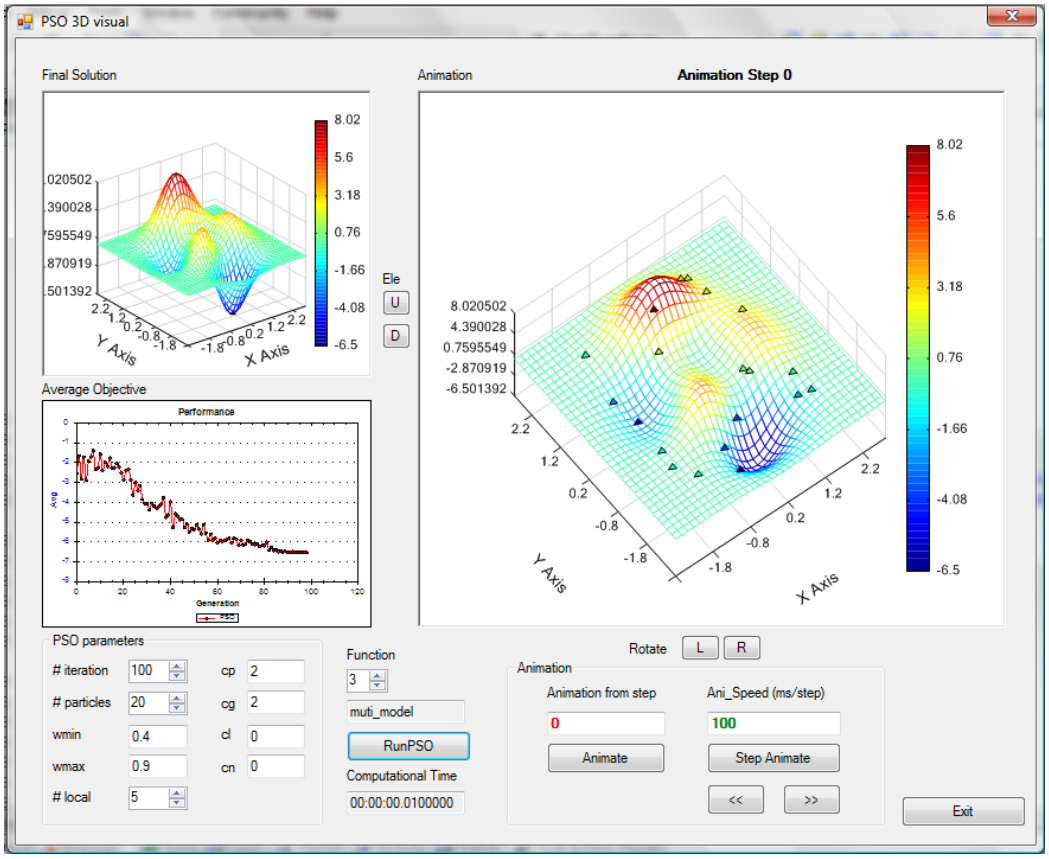

Figure 2.6: 3D Visual presentation of GLNPSO algorithm

2.2. GLNPSO components

In the remainder of this chapter, the structure of GLNPSO provided in ET-library is discussed in detail. In Figure 2.7, the class view of GLNPSO is presented. Generally, there are three important classes required for GLNPSO: Particle, Swarm and PSO.

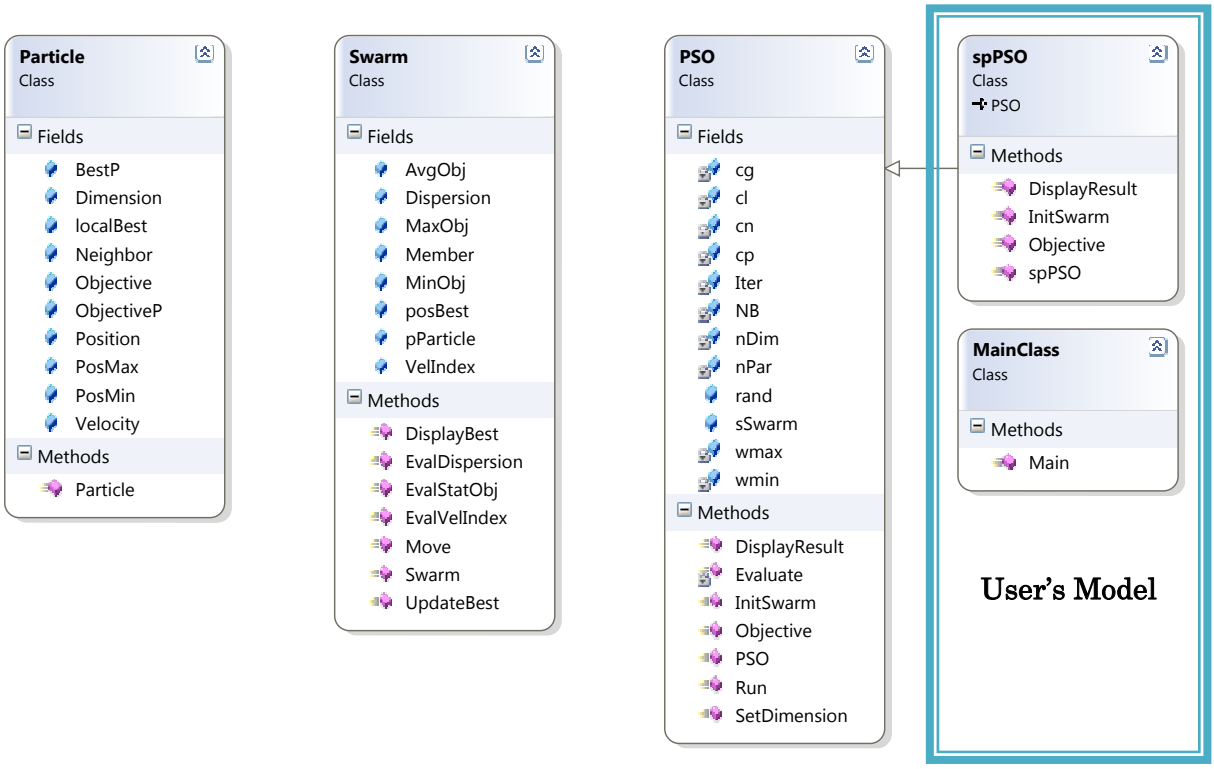

Figure 2.7: Class view for GLNPSO in ET-library

2.2.1. Particle class

Particle is the basic class in ET-Lib which includes all the information related to a particular particle. Here are the definitions of attributes of a particle:

Name	Type	Description
Position	Array of real number	m-dimension position of the particle
Dimension	Integer	the dimension of particles' positions
PosMin/PosMax	Array of real number	the lower and upper bounds of position of at each dimension
Velocity	Array of real number	m-dimension velocity of the particle
Objective	Real	the objective value or the fitness of the particle
BestP	Array of real number	m-dimension position of the particle which stores its personal best experience
ObjectiveP	Real	the objective value corresponding to BestP

localBest	Integer	the index (or location) of local best member in the swarm
Neighbor	Array of Real	m-dimension position which is identified by comparing the relative position and objective and position of the particle with other members in the swarm

The constructor `public Particle(int nDim)` is used to create a new particle. The parameter `nDim` indicates the Dimension of a particle.

2.2.2. Swarm class

A swarm is consisted of many particles flying in the search space to look for good position. The swarm class includes all the required routines to govern the movement behavior of its members (particles). The attributes and methods of this class are listed below:

Attributes

Name	Type	Description
Member	Integer	number of particles in the swarm (population size)
pParticle	Array of Particle	a set of particles in the swarm
MinObj/MaxObj	Real	the minimal and maximal objective value found by the swarm through searching process
AvgObj	Real	the average objective values across all particles in the swarm
postBest	Integer	the index (or location) of global best member in the swarm (pParticle[postBest] refers to the particle which found the position resulting in the best objective value)
VelIndex	Real	the velocity index to measure how fast the swarm is moving
Dispersion	Real	the Dispersion index to measure the dispersion of particles in the swarm

Methods

`public Swarm(int nPar, int nDim)`	create a new swarm by determine the number of particles in the swarm (`nPar`)and the Dimension of each particle)
`public void DisplayBest()`	show the information of the global best particle on the screen
`public void Move(double w, double cp, double cg, double cl, double cn, double[,] r1, double[,] r2, double[,] r3, double[,] r4)`	calculate velocities of particles and move them to new positions. The parameters passed to this method include the inertia weight, acceleration constants and random numbers.
`public void UpdateBest(int nbSize)`	update information related to personal best, global best, local best and neighbor best after each flying attempt.
`public void EvalStatObj()`	update statistics related to the objective values of particles in the swarm.
`public void EvalDispersion()`	evaluate Dispersion index
`public void EvalVelIndex()`	evaluate Velocity index

2.2.3. PSO class

All the PSO parameters and routines are stored in this class. Some methods in this class are problem-oriented and can be overridden when formulating new optimization problems. In general, it has following attributes and methods:

Attributes:

Name	Type	Description
$c_p/c_g/c_l/c_n$	Real	personal/global/local/neighbor acceleration constant
`Iter`	Integer	number of iterations
`NB`	Integer	number of neighbor
`nDim`	Integer	dimension of particles in a swarm
`nPar`	Integer	Number of particles
`Rand`	random stream	random object used to generate random number
`sSwarm`	Swarm	the swarm used in the PSO algorithm
`wmax/wmin`	Real	the maximal/minimal inertia weight (normally the inertia weight in our the default GLNPSO is linearly reduced at each iteration from wmax to wmin)

Methods

`public virtual void DisplayResult(TextWriter t)`	write the results of GLNPSO to a predefined output file `t`
`public virtual double Objective(Particle p)`	objective function
`void Evaluate()`	`Objective(Particle p)` is called to evaluate the objective value of each particle in the sSwarm
`public virtual void InitSwarm()`	initialize sSwarm with random particles
`public PSO(int nIter, int nNB, double dwmax, double dwmin, double dcp, double dcg, double dcl, double dcn)`	create a new PSO object by passing all PSO parameters
`public void Run(TextWriter t, bool debug)`	perform GLNPSO algorithm
`public void SetDimension(int par, int dim)`	set swarm size and particle's dimension

The GLNPSO algorithm is implemented in *Run* method. The basic framework of this algorithm is similar to that of the algorithm introduced in section 1.4. The algorithm first initialize new swarm with user's predefined parameters such as number of particle, and dimension of a particle. After a random swarm is created, their fitness (objective value) is evaluated and the learning terms are updated. Then, the swarm starts to evolve until the stopping condition is met. The dispersion index and statistics collections routines can be called optionally. The C# implementation of this algorithm is presented in Figure 2.8.

When designing this library, our objective is to minimize the users' effort to rewrite the PSO algorithm. To solve an optimization problem with GLNPSO, the users only need to focus on objective function evaluation procedure (encoding/decoding approach) to make the program faster and more effective in finding high quality solutions. For easy problem such as the first example, a simple class defined in Figure 2.2 is all the user needs to create to use GLNPSO in ET-library. For more complicated problems, some modifications in GLNPSO routines such as movement strategies, local search, and re-initialization may be added as required. In the next chapter, we introduce some practical applications of GLNPSO and also show the flexibility of the design.

```
public void Run(TextWriter t, bool debug)
{
   //PSO main iteration
   double w = wmax;
   double decr = (wmax - wmin) / Iter;
   sSwarm = new Swarm(nPar, nDim);
   InitSwarm();
   Evaluate();
   sSwarm.UpdateBest(NB);

   if (debug)
   {
      sSwarm.EvalDispersion();
      sSwarm.EvalStatObj();
   }
   for (int i = 1; i < Iter; i++)
   {
      ## Generate random number u1, u2, u3, u4 ##
      sSwarm.Move(w, cp, cg, cl, cn, u1, u2, u3, u4);
      Evaluate();
      sSwarm.UpdateBest(NB);
      if (debug)
      {
         sSwarm.EvalDispersion();
         sSwarm.EvalStatObj();
      }
      w -= decr;
   }
}
```

*** the code in ## ... ## contain the subroutine which can be found in the original code*

Figure 2.8: C# implementation of GLNPSO algorithm

CHAPTER 3

GLNPSO's APPLICATIONS

3.1.Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) is a traditional problem which is normally used as a benchmark for many optimization methods. In TSP, a list of locations is given and the task is to find the tour that minimizes the total distance through all locations provided that each location can only be visited once.

For instance, the salesman begins his tour at location 0 and need to visit N location before coming back to the starting location. Our objective is to find the shortest path $\pi = \{l_1, l_2, \dots, l_N\}$ for this task given that no location will be revisited (except for location 0). An example of a TSP's solution is shown in Figure 3.1. With a set of predetermined locations, a N+1 by N+1 distance matrix D is defined and the distance, time, or the cost to travel from location i to location j is defined by $D[i, j]$. The mathematical model of this problem can be simply:

$$Minimize \quad D[0, l_1] + \sum_{k=1}^{N-1} D[l_k, l_{k+1}] + D[l_N, 0] \ , with\ l_k\ is\ the\ location\ index \qquad (3.1)$$

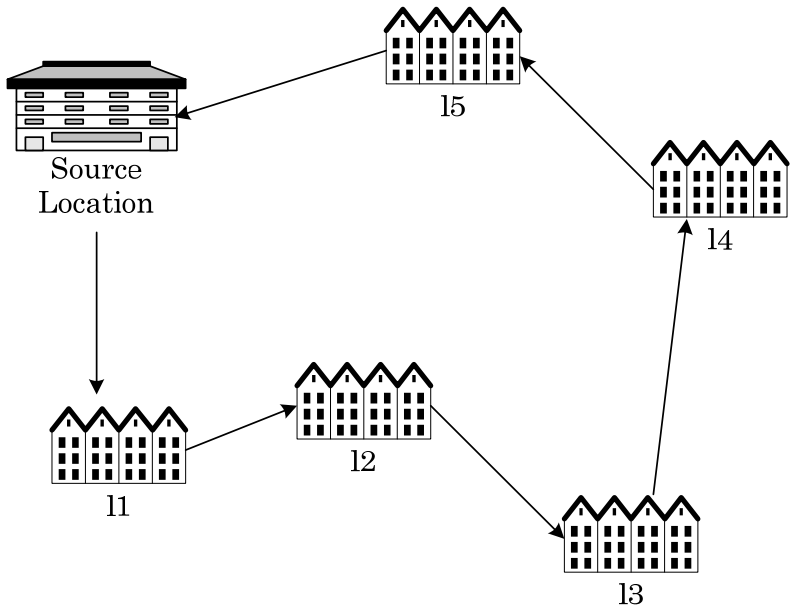

Figure 3.1: TSP solution $\pi = \{l_1, l_2, \dots, l_5\}$

It is well known that TSP is in the class of NP-complete problems that the computational time to find the optimal solution increase exponentially with the number of locations. For that reason, a lot of heuristic approaches have been proposed for this problem. In this section, the TSP is solved by using the GLNPSO algorithm. Different from the example in chapter 2, the particle's positions of TSP cannot be directly used to calculate the objective value (total distance). Instead, particles must be decoded to get TSP' solutions. The encoding/decoding scheme is presented in Figure 3.2.

Location	1	2	3	4	5
Particle's position	0.123	0.234	0.04	0.523	0.421

↓

Traveling route π	3	1	2	5	4
	0.04	0.123	0.234	0.421	0.523

Evaluate Objective value

$$D[0,l_1] + \sum_{k=1}^{4} D[l_k, l_{k+1}] + D[l_5, 0]$$

$$= D[0,3] + D[3,1] + D[1,2] + D[2,5] + D[5,4] + D[4,0]$$

Figure 3.2: Encoding/Decoding approach for TSP with N=5

In figure 3.2, the position of a particle is an array of real number randomly distributed from 0 to 1. Each position in the m-dimension position is used to indicate the priority of a location. The locations with smaller position values will be visited before those with larger position values. In the decoding method, the traveling is determined by sorting the particle's position in the ascending order. When the route π has been constructed, the total distance of the tour is calculated.

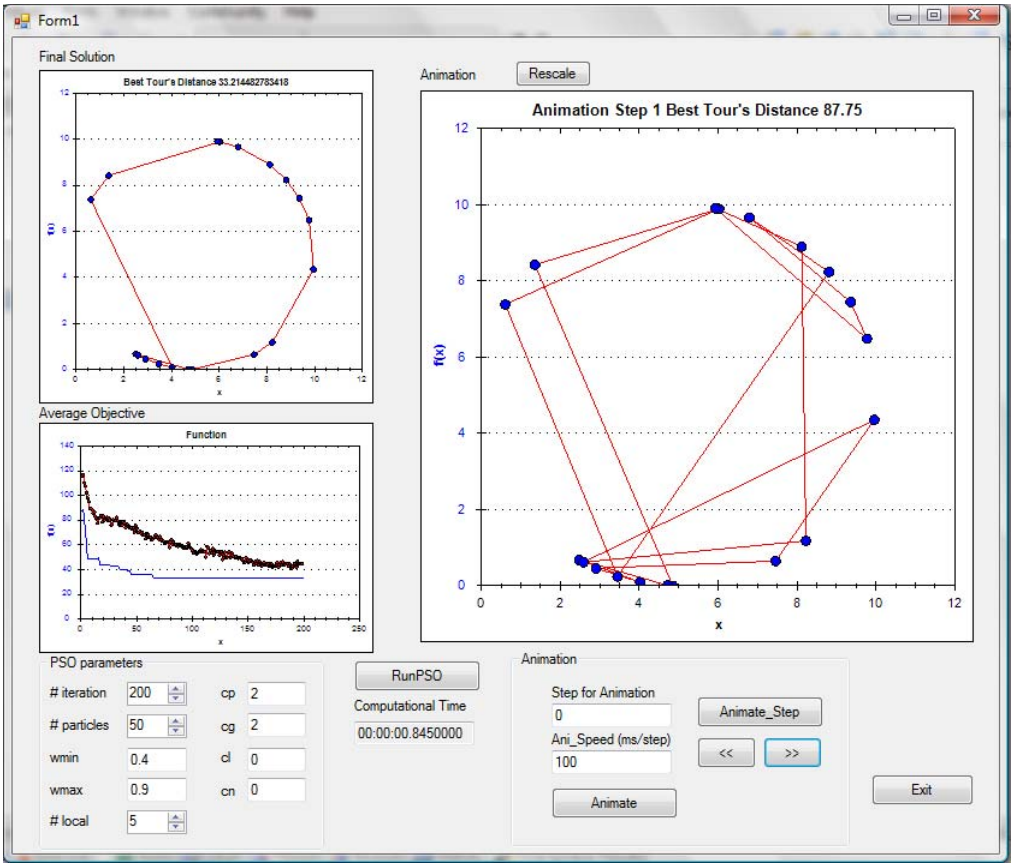

Figure 3.3: TSP optimizer with GLNPSO library

At each iteration, the fitness (objective value) of each particle in the swarm is evaluated by this decoding procedure. The user can find the source code of the application in Figure 3.3 at "\GLNPSO basic\Applications\PSO_Visual_TSP\". The coordinate of each location is in the file "\GLNPSO basic\Applications\PSO_Visual_TSP\PSO basic_visual_TSP\bin\DebugLocations.txt" and the format of this file is:

Number of locations
For each location: x-coordinate, y-coordinate

3.2.Job Shop Scheduling (JSP)

The job shop scheduling problem (JSP) is a combinatorial optimization problem in which a set of jobs need to be scheduled on a set of machines in order to optimize a certain criterion followed by the constraints that each job has the precedence and deterministic time-span which are known in advance. Each sequencing job that consists of **n** operations will be processed on a set of **m** machines; hence, there are a total of **nm** activities (operations) involved in such a job shop scheduling problem. In Table 3.1, an example of JSP with 4 jobs and 3 machines are given and a feasible solution of this problem is illustrated in Figure 3.4.

Table 3.1 The 4×3 example of JSP

Job	Machine sequence			Processing Time		
1	M1	M2	M3	3	2	8
2	M2	M1	M3	5	3	7
3	M3	M1	M2	4	1	5
4	M1	M3	M2	9	7	1

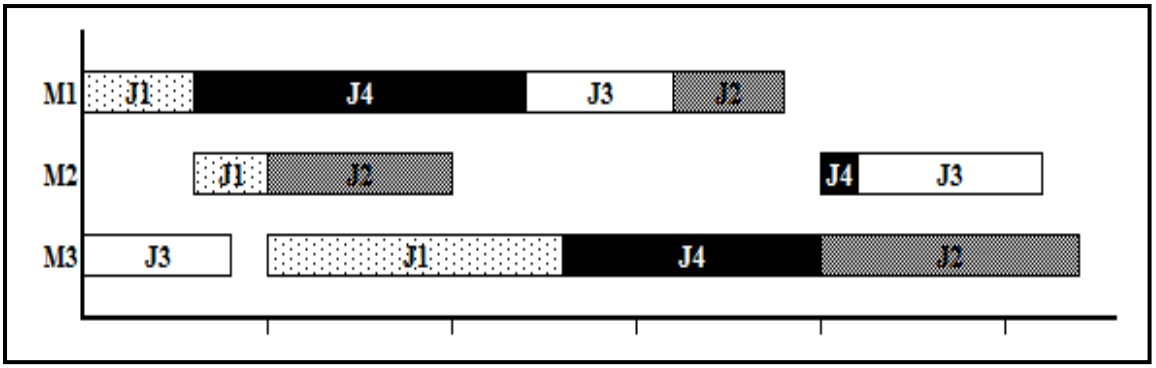

Figure 3.4: Feasible solution for a job shop scheduling problem

Small size instances of the JSP can be solved within reasonable computational time by exact algorithms. However, when the size of problem is increased, the computational time of the exact approaches grow exponentially. Accordingly, many researchers develop heuristic techniques to achieve near optimal solution instead. Nevertheless, the heuristic approaches are problem specific and they might not be applicable to all situations; thus, meta-heuristics are investigated to improve the quality of the solution as well as increase the computational speed.

3.2.1. JSP's model

In this section, we will create a model of this problem with GLNPSO to minimize the makespan C_{max} (maximum completion time of all operations). Following is the mathematical model of JSP:

Notations in the JSP

Indices:

j : The j^{th} job in the problem, $j = \{1,...,n\}$

k : The k^{th} machine in the problem, $k = \{1,...,m\}$

Decision variable:

$x_{j,k}$: The start time of job j on machine k.

$y_{j,j',k}$: $\begin{cases} 1 & \text{if job } j \text{ is scheduled before job } j' \text{ on machine } k. \\ 0 & \text{Otherwise.} \end{cases}$

Parameters:

m : The number of machines.

n : The number of jobs.

$p_{j,k}$: The process time of job j on machine k.

r_j : The ready time of job j.

d_j : The due date of job j.

M : An arbitrary large number.

Objectives:

The objective functions are frequently to minimize any of the performance measures as the following. Some commonly used objectives in the JSP include the followings;

- Minimize: $\max_{j,k} \{ x_{j,k} + p_{j,k} \}$

Subject to:

Precedence constraints	$x_{j,k} + p_{j,k} \le x_{j,k'}$	$\forall j,k,k'$	(3.2)
Conflict constraints	$x_{j,k} + p_{j,k} \le x_{j',k} + M(1 - y_{j,j',k})$	$\forall j,j',k$	(3.3)
	$x_{j',k} + p_{j',k} \le x_{j,k} + M\, y_{j,j',k}$	$\forall j,j',k$	(3.4)
Readiness constraints	$x_{j,k} \ge r_j$	$\forall j,k$	(3.5)
Nonnegative constraints	$x_{j,k} \ge 0$	$\forall j,k$	(3.6)
	$y_{j,j',k}$ *binary*	$\forall j,j',k$	(3.7)

3.2.2. Encoding/Decoding

It is obvious that the solutions for JSP cannot be directly represented as the m-dimension position as introduced in chapter 2. For that reason, we will use encoding/decoding method so as to the solutions of this problem can be expressed as particles' positions which are evolved through PSO algorithm. Then, the position is decoded to get the feasible solution to evaluate the objective value. In this example, we

use an array of real numbers to represent the priority of each operation that needs to be scheduled. The schematic illustration of this encoding/decoding procedure for JSP in table 3.1 is shown in Figure 3.5.

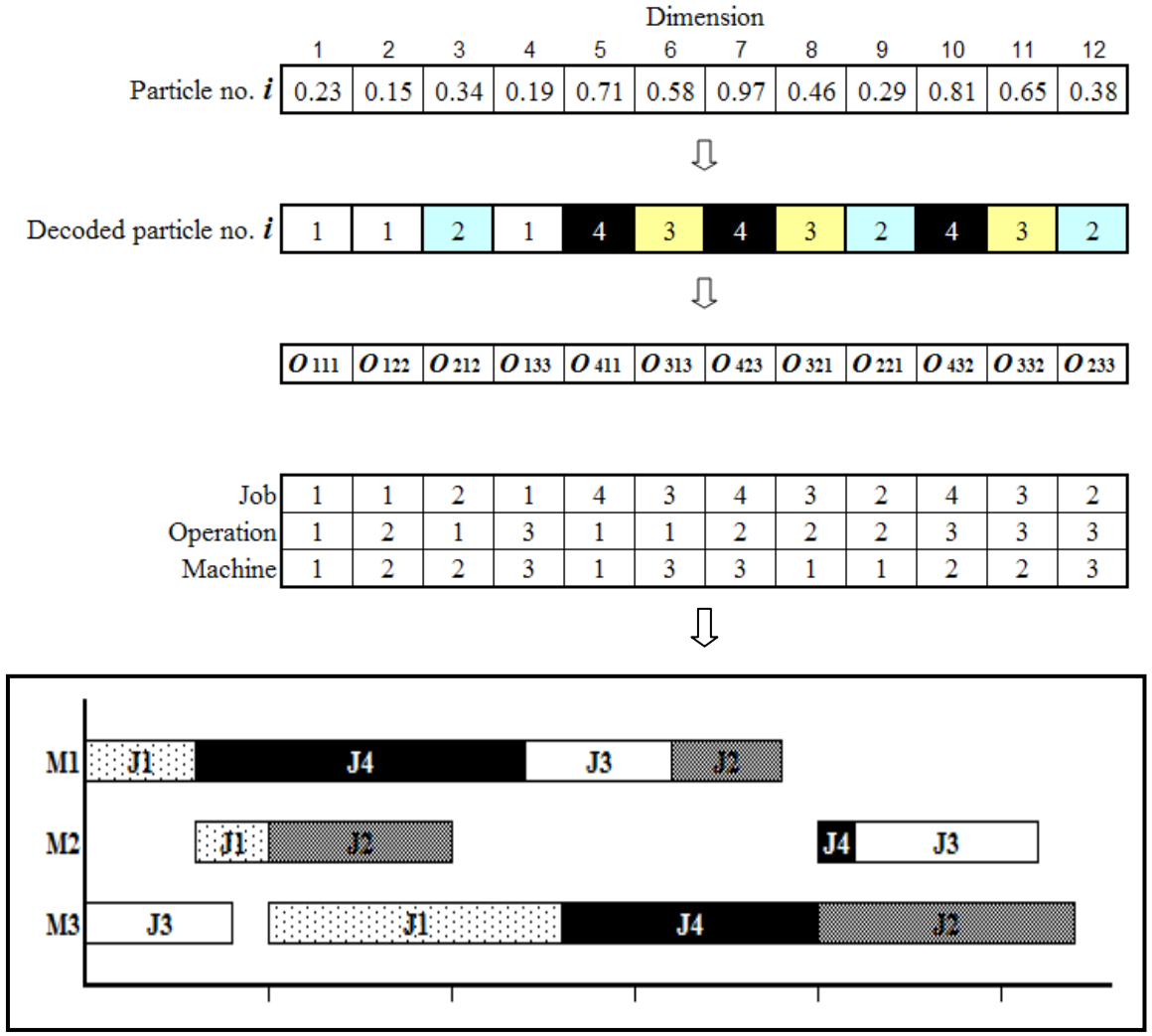

Figure 3.5: Encoding/Decoding procedure
with the operation-based representation of a particle

First, each solution is encoded in a particle's position as an array of real numbers which are randomly generated in range [0, 1]. The dimension of each particle equals to the number of jobs multiplied by the number of machines. In this example as shown in Table 3.1, there are 4 jobs and 3 machines; thus, the dimension of particle for this example equals to 12.

At each step in PSO algorithm, particles are decoded to get feasible schedules. The m-repetition of job numbers permutation which was first introduced by Tasgetiren et al. (2005) is applied along with sorting list rule. Firstly, the continuous numbers inside particle will be sort then the permutation of 3-repetition of 4 jobs will be applied. After that, the operation-based approach by Cheng *et al.* (1996) is used to represent a schedule. The advantage of this approach is that any permutation of this representation always leads to a feasible schedule. Nevertheless, it is possible that some of different representations could possibly generate the same schedule. The particle as shown previously is used, corresponding to the small size of JSP which is already mentioned.

The decoded particle is then transferred to a schedule by taking the first operation from the list, the second and so on. During the schedule generation, each operation is allocated to a required machine in the best available processing time without delaying other scheduled operation. The procedure yields an active schedule. For instance O_{122} (Job 1, Operation 2, Machine 2) is allocated to the machine 2 at time 3. It cannot be scheduled before time 3 because the first operation of Job 1 is being processed.

The source code for JSP with GLNPSO can be found in the manual folder, which mainly based on PSO algorithm proposed by Pratchayaborirak (2007). The main different between this structure of this model and that of the example in chapter 2 is the introduction of some specific classes to store data of JSP and perform decoding procedure and evaluate objective value (except makespan, several other objective values can be easily calculated after particles are decoded). The general view of this model is given in Figure 3.6.

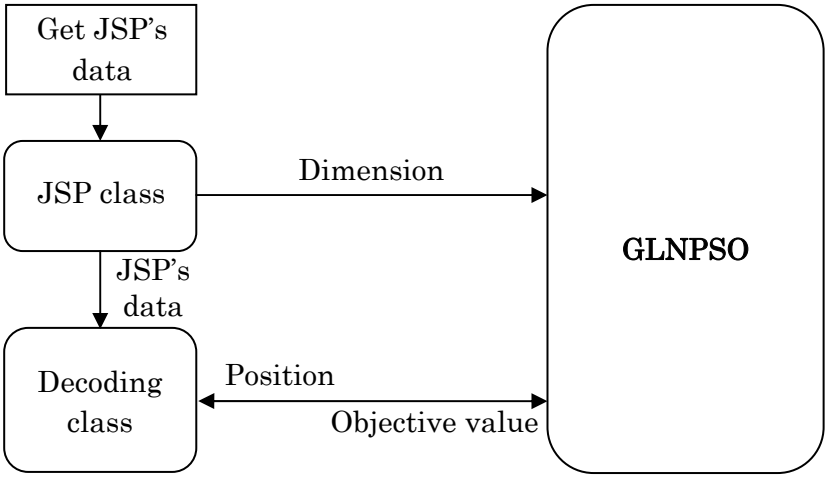

Figure 3.6: GLNPSO model for JSP

3.2.3. Reinitialize strategy

During the iterations, the particles are often trapped in a deep local minimum which can cause trivial movement of the whole swarm. As a result, the reinitialize strategy is applied to diversify the particles over the search space once again. Consequently, the system could escape from that local trap. This approach can be applied to enhance PSO as shown below.

Suppose that the algorithm met the re-initialize criteria which has been set in advance then the re-initialize algorithm will start when the certain iteration number is reached and the procedure will be repeated again every fixed number of iteration. To accomplish the re-initialize strategy, a pre-defined number of particles are randomly selected for re-initialization. This number is defined by the reinitialized ratio multiply by the number of particles. In addition, the *gbest* particle is excluded from the selection. The personal memories of each selected particle are reinitialized by randomly regenerating its position, resetting its velocity to zero, and resetting *pbest* to null.

3.2.4. Local search strategy

In general, a local search may apply to a certain group of particles in the swarm to enhance the exploitation of search space. The local search typically attempts to improve quality of the solution by searching the better solutions around its neighbors. In this study, the neighborhood search adopts the critical block (CB) neighborhood of Yamada and Nakano (1995). Concept of the search method is to move an operation inside a critical block to the beginning or the end of that critical block. Figure 3.7 presents the critical path and the set of neighborhood move according to the CB neighborhood.

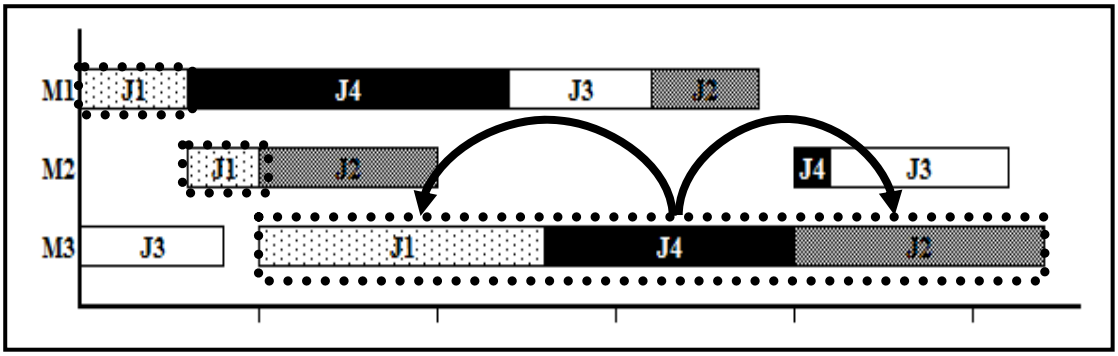

Figure 3.7: The CB Neighborhood

The local search procedure used in the propose algorithm can be described as the following.

Suppose the algorithm meet the local search criteria which already set in advance then the local search algorithm will be activated by the reaching of a certain iteration number and the local search procedure will repeat every fixed number of iteration.

To perform the local search, firstly, a critical path – the path with the longest length from the first operation on any machine to the last operation on any machine – is identified. A single critical path is arbitrarily selected if there is more than one critical path. Any operation on the critical path is called a critical operation. The critical path is naturally decomposed into critical blocks. The block is a maximal subsequence of critical operations that are processed on the same machine. Therefore, two consecutive blocks require different machines to process those operations.

A moving set of neighborhood is defined inside the block which contains at least three operations, any operation between the first and last operation in a critical block is moved to the beginning or the end of that critical block. Furthermore, a block which contains two operations, two operations will be simply swapped.

For each move according to the defined set, if the fitness value is improved then the new solution and the new fitness value are updated. The local search procedure ends when all moves are completed.

The reinitialize and local search strategy are added to the original algorithm in order to improve the quality of final solutions by making some attempts to escape from the local optimal. `LocalSearchParticle(sSwarm.pParticle[j], ref rand)` and `ReInitSwarm()` are new methods in PSO class. At the beginning of an iteration, if the reinitialize or local search condition are met, the swarm will respectively reinitialize or perform local search on its members instead of performing movement. Following is the C# implementation of GLNPSO for JSP.

```
public void Run(TextWriter t, bool debug)
{
   //PSO main iteration
   double w = wmax;
   double decr = (wmax - wmin) / Iter;
   sSwarm = new Swarm(nPar, nDim);
   InitSwarm();
   Evaluate();
   sSwarm.UpdateBest(NB);

   if (debug)
   {
      sSwarm.EvalDispersion();
      sSwarm.EvalStatObj();
   }
   for (int i = 1; i < Iter; i++)
   {
      bool reinit_locals = false;
      if (((i - startLS) % LSiterval == 0) && (i >= startLS))
      {
       for (int j=0; j<sSwarm.Member; j++)
            LocalSearchParticle(sSwarm.pParticle[j], ref rand);
       reinit_locals = true;
      }
      if (((i - startReinit) % ReInitIterval == 0) && (i >= startReinit))
      {
       ReInitSwarm();
       reinit_locals = true;
      }
      if (!reinit_locals)
      {
       ## Generate random number u1, u2, u3, u4 ##
       sSwarm.Move(w, cp, cg, cl, cn, u1, u2, u3, u4);
      }
      Evaluate();
      sSwarm.UpdateBest(NB);
      if (debug)
      {
         sSwarm.EvalDispersion();
         sSwarm.EvalStatObj();
      }
      w -= decr;
   }
}
```

Call reinitialize and local search

*** the code in ## ... ## contain the subroutine which can be found in the original code*

Figure 3.8: C# implementation of GLNPSO algorithm for JSP

3.2.5. Migration strategy

After a swarm met the stopping criteria, some particles will migrate to the next swarm, with random number, equal to the number of migrating particles which already set in advance as a percentage of migration. The migration strategy can also diversify the particles over the search space again. Consequently, the solution may be improved by exploring new area in the search space and exploiting the good flying experience from migrated particles.

Pratchayaborirak (2007) used this concept in his two-stage PSO algorithm. The first stage of the algorithm consists of **k** swarms which are serially executed using the same objective function. When a certain swarm is terminated, a percentage of particles will be randomly selected to migrate to the next swarm to join with the newly generated particles. This can help boost the convergence of solution by using information from the previous swarm. The first stage ends when the fourth swarm is terminated.

In the second stage, equal numbers of particles are randomly selected from the four previous swarms to form a single swarm and the PSO algorithm is repeated until the stopping condition is met. The best result yields at the end of the second stage will be used as the best answer found. The two-stage PSO algorithm is performed in the Main class as shown in Figure 3.9.

```
class MainClass
{
public static void Main(string[] args)
{
## Read input from file ##
JD ← ## calculate dimension of particles based on JSP data ##

int noPar = 10;
int noIter = 200;
int noNB = 5;
double wMax = 0.9;
double wMin = 0.4;
double cP = 2;
double cG = 2;
double cL = 0;
double cN = 0;
string oFile = "MyPSO.xls";

double MigrateProp = 0.2;
bool multiSwarm = true;
int noSwarm = 5;

int startReinit = 150;
int ReInitIterval = 100;
int startLS = 210;
int LSinterval = 100;

int noRep = 3
// starting time and finish time using DateTime datatype
DateTime start, finish;
// elapsed time using TimeSpan datatype
TimeSpan elapsed;
## opening output file ##
for(int i=0; i<noRep; i++)
{
  Console.WriteLine("Replication {0}", i+1);
  tw.WriteLine("Replication {0}", i+1);
  // get the starting time from CPU clock
  start = DateTime.Now;

  // main program ...
  PSO[]  subSwarm=new PSO[noSwarm-1];
  #region Activate sub-swarms
  if (multiSwarm)
  {
```

parameters for two-stage PSO algorithm

Parameters for local search and re-initialize strategy

```
    for (int s = 0; s < noSwarm - 1; s++)
    {
      Console.WriteLine("Start swarm {0}", s);
      subSwarm[s] = new spPSO(noPar, noIter, noNB, wMax, wMin, cP, cG, cL,
                                          cN, Dimension, JD,ReInitIterval);
      if (s != 0) subSwarm[s].Migrate(subSwarm[s - 1].sSwarm,
                                          subSwarm[s].sSwarm, MigrateProp);
      subSwarm[s].Run(tw, true);
       subSwarm[s].DisplayResult(tw);
       Console.WriteLine("Obj {0} ",
subSwarm[s].sSwarm.pParticle[subSwarm[s].sSwarm.posBest].ObjectiveP[0]);
     }
    }
  #endregion
  Console.WriteLine("Start final swarm");
  PSO globalSwarm = new spPSO(noPar, noIter, noNB, wMax, wMin, cP, cG, cL,
                                        cN, Dimension, JD, ReInitIterval);
  if (multiSwarm)
   {
      for (int s = 0; s < noSwarm - 1; s++)
         globalSwarm.MigrateBest(subSwarm[s].sSwarm, globalSwarm.sSwarm, 1
                                                / ((double)noSwarm - 1));
   }
  globalSwarm.Run(tw, true);
  globalSwarm.DisplayResult(tw);
## display results ##
}
tw.Close();
}
}
```

Migrate and evole sub-swarm

Collect best members in previous swarms evolve a global swarm

*** the code in ## ... ## contain the subroutine which can be found in the original code*

Figure 3.9: C# implementation of two-stage PSO algorithm

In Figure 3.9, `subSwarm[s].Migrate(subSwarm[s - 1].sSwarm, subSwarm[s].sSwarm, MigrateProp)` is a new method in Swarm class to randomly migrate a proportion of particles from one swarm to another. On the other hand, `globalSwarm.MigrateBest(subSwarm[s].sSwarm, globalSwarm.sSwarm, 1 / ((double)noSwarm - 1))` is performed to equally collect top members in the sub-swarms into a global swarm. The details of these methods are presented in source code.

The coordinate of each location is in the file "\GLNPSO basic\PSO JSP\PSO basic\bin\Debug\JSP.txt" and the format of this file is:

Number of jobs, number of machines
For each job:
    For each operation: machine ID, processing time

CHAPTER 4

MULTI-OBJECTIVE OPTIMIZATION WITH PSO

Previous chapters have shown how GLNPSO can be used to solve optimization problems with single objective. However, many real world applications required optimization models to handle more than one objective function. As a result, multi-objective optimization (MO) becomes increasingly attractive to both practitioners and researchers. So far, there have been a large number of studies focusing on methodologies to deals simultaneously with more than one objective function. The mathematical model for a MO problem is given as follow:

$$minimize \qquad \vec{f}(\vec{x}) = [f_1(\vec{x}), f_2(\vec{x}), \dots, f_K(\vec{x})] \tag{4.1}$$

subject to:

$$g_i(\vec{x}) \leq 0 \;\; i = 1,2, \dots m \tag{4.2}$$

$$h_i(\vec{x}) = 0 \;\; i = 1,2, \dots l \tag{4.3}$$

where $\vec{x}$ is the vector of decision variables, $f_i(\vec{x})$ is a function of $\vec{x}$, K is the number of objective function to be minimzed, $g_i(\vec{x})$ and $h_i(\vec{x})$ are the constraint functions of the problem.

4.1.Review of methodologies for multi-objective optimization

One of the most intuitive methods to solve multi-objective problem is to combine the objectives into a single aggregated objective function. In this method, each objective function will be assigned a weight based on the preference of the decision makers and all of these weighted functions are linearly combined. The only remaining task is to use any available optimizer to find the solution for the problem with this single aggregated objective function. However, this approach has two major drawbacks. Firstly, a single solution is obtained based on a set of pre-defined, subjective weights on the objective functions. Thus the requirement of prior preference of the decision makers may not lead to a satisfactory result (another approach based on prior preference is goal programming which normally solve MO problem as a series of linear programs). Secondly, the decision maker's knowledge about the range of each objective value may be limited. As a result, even with a preference in mind, the single solution obtained provides no possibility for tradeoffs of decisions. In order to be more objective, the approach based on a single aggregative objective function needs to be run multiple times to see the effect of the weights on the solutions obtained. Hence it is more preferable to provide means for the decision maker to find the tradeoff by identifying the non-dominated solutions or Pareto front, which usually consumes a relatively large amount of computational time. For that reason, many methods are developed to search for the Pareto front. In this case, multi-objective Evolutionary Algorithm (EA) is the most commonly selected solution technique.

One of the earlier attempts to solve multi-objective optimization problems using Evolutionary Algorithm (MOEA) is Non-dominated Sorting Genetic Algorithm or NSGA (Srinivas and Deb, 1995). This method was commonly criticized for its high

computational complexity which made it inefficient with a large population size. Another problem with this method is that its effectiveness depends mostly on the pre-defined sharing parameter. To address the drawbacks of the original NSGA, the new NSGA-II is proposed (Deb et al., 2002) by adopting a new non-dominated sorting procedure, an elitism structure, and a measurement of crowdedness. In their paper, NSGA-II had been demonstrated to outperform other MOEAs such as Pareto-archived evolutionary strategy (PAES) and strength- Pareto EA (SPEA).

4.2.Pareto Optimality

For the formulation 4.1-4.3, given two decision vectors $\vec{x}, \vec{y} \in R^D$, the vector $\vec{x}$ is considered to dominate vector $\vec{y}$ (denote $\vec{x} \prec \vec{y}$), $if\ f_i(\vec{x}) \leq f_i(\vec{y})\ for\ \forall i = 1,2,\dots,K\ and\ \exists j = 1,2,\dots K | f_j(\vec{x}) < f_j(\vec{y})$.

As shown Figure 4.1, for the cases that neither $\vec{x} \prec \vec{y}$ nor $\vec{y} \prec \vec{x}$, $\vec{x}\ and\ \vec{y}$ are called non-dominated solutions or "trade-off" solutions. A non-dominated front $\mathcal{N}$ is defined as a set of non-dominated solutions if $\forall x \in \mathcal{N}, \nexists y \in \mathcal{N} | \vec{y} \prec \vec{x}$. A Pareto Optimal front $\mathcal{P}$ is a non-dominated front which includes all solution $\vec{x}$ non-dominated by any other $\vec{y} \in \mathcal{F}, \vec{y} \neq \vec{x}$ where $\mathcal{F} \in R^D$ is the feasible region.

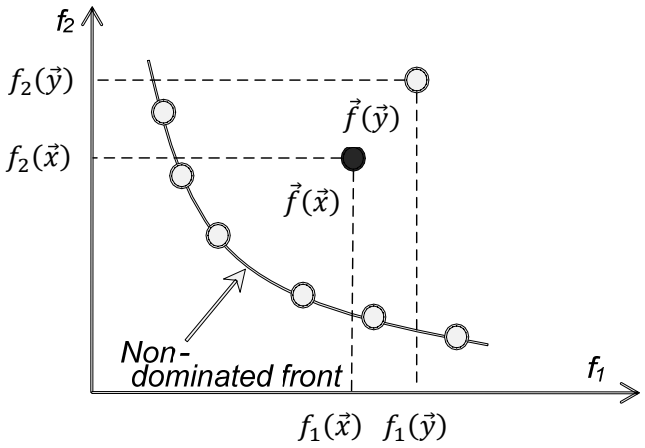

Figure 4.1: $\vec{x} \prec \vec{y}$ for the case with two objective functions

4.3.Multi-objective optimization with PSO

As discussed earlier, one of the approaches for solving problems with multiple conflicting objective functions is to search for Pareto optimal front, i.e., to search for the set of non-dominated solutions. This Pareto optimal front represents the best solution for the problems with multiple conflicting objective functions. It is quite a different proposition from searching for a single best point and it is necessary to modify the original framework of PSO. The key components to be modified include the following:

- Storage of elite group or non-dominated solutions found so far
- Selection of a reference particles (or leaders) to guide the swarm toward better positions
- Movement strategy, how to use the reference particles as search guidance

In the multi-objective optimization problems, the flying experience of the swarm needs to be stored as a set of non-dominated solutions instead of a single solution. In this case, the Elitist structure as mentioned in NSGA-II is adopted. After each update of particle position, the objective functions of each particle are evaluated and they must all be processed by a non-dominated sorting procedure. This sorting procedure identifies the group of particles in the swarm which are non-dominated by other particles and put all of these particles into an archive for the Elite group. Again, the Elite group is screened to eliminate inferior solutions, i.e., solutions that were dominated by those in the Elite group. As a result, the Elite group in the archive is the best non-dominated solutions found so far in the searching process of the swarm.

When the Elite group is formed, one of the biggest challenges for most EAs is how to select the candidates among the Elite group to help guide the evolution of the population. The most common criterion is that the leader (or guidance) needs to lead the population to the less crowded areas to obtain a better spread of the final front. A successful implementation of this idea is given in NSGA-II with the introduction of crowding distance (CD) as a measure of the spread of the non-dominated front. This approach estimates the density of solutions surrounding a specific solution by calculating the average distance of two points on either side of this point along each of the objective (see Deb et al., 2002 for more details). The advantage of this approach is that it does not require a pre-determined sharing parameter in NSGA. Coello el al., 2002 proposed a PSO algorithm with a geographically-based system to locate crowded regions. They divided the objective space into a number of hypercubes and then each member in the Elite archive is assigned to one of these hypercubes. After the archive is classified, a hypercube with smallest density is considered and one of its members is randomly selected to be used as the global guidance.

Finally, the movement of particles is very critical to improve the quality of the Pareto front. Most of the proposed Multi-objective PSO (MOPSO) algorithms use only a single global guidance from the Elite group similar to the traditional PSO movement strategy. However, the existence of multiple candidates in the archive may open a large number of choices for movements. In section 4.4, several potential movement strategies are discussed as options to fully utilize the Elite archive as guidance for the search.

In Figure 4.2, a PSO framework for multi-objective optimization problems is presented. This framework takes into account all the features that are mentioned above and the implementation of this framework is described in algorithm A1. The $Non\_dominated\_Sort$ ($\mathcal{S}$) uses the sorting algorithm proposed in NSGA-II to identify non-dominated solutions. After each particle is evaluated, the set of non-dominated solutions will be updated and stored in the Elite group. The number of solutions in the Elite group is usually limited to reduce the computational time for sorting and updating procedures. When the number of non-dominated solutions exceeds the limit, the particles located in the crowded areas will be selectively removed, so the Elite group can still result in a good Pareto front. The two procedures $Select\_Guidance$ ($\mathcal{E}$) and $Update\_velocity(g)$ are movement strategy dependent and will be separately discussed in the next section.

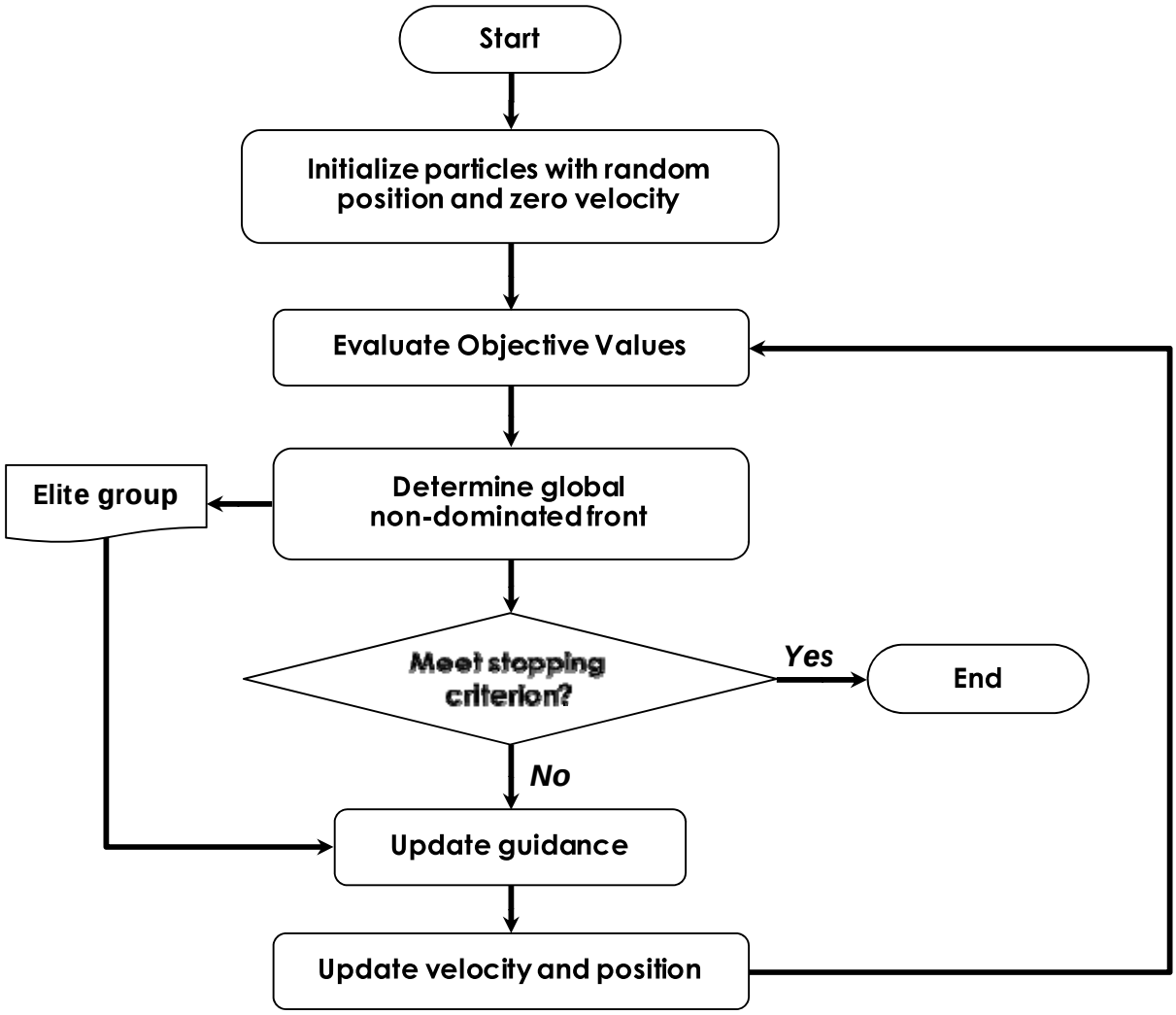

Figure. 4.2: Framework for MOPSO

A1. Algorithm for MOPSO

i. *Initialize the swarm $\mathcal{S}$ and set the velocities of all particle to zero*

ii. *For each particle $i \in \mathcal{S}$ with position Θ_i*

$$\textit{Evaluate objective function } f_k(\Theta_i), \forall k = 1,2,\dots,K$$

iii. $\mathcal{S}^* \leftarrow Non\_dominated\_Sort\ (\mathcal{S})$ *- $\mathcal{S}^*$ is the set of non-dominated particles in $\mathcal{S}$*

iv. *Elite archive* $\mathcal{E} \leftarrow Non\_dominated\_Sort\ (\mathcal{E} \cup \mathcal{S}^*)$

v. *If the stopping criterion is satisfied, end procedure; otherwise, go to step vi*

vi. *Update_social _learning _terms*

vii. *Global guidance* $g \leftarrow Select\_Guidance\ (\mathcal{E})$

viii. *Update_velocity*(g) *using equation (1.10)*

ix. *Update_position by equation (1.11)*

x. *Return to step ii*

In this framework, the multiple social learning terms in GLNPSO are used to update the new velocity. As a result, the new velocity is influenced by four social terms: personal best, global best (global guidance), local best and near neighbor best. The global guidance is the most important term in this framework and it depends mainly on the movement strategy adopted by the swarm; therefore it will be discussed separately. The modifications for other terms are adjusted in this framework to make it work for MO problems.

In MO problems, there are two situations when the personal best need to be updated. First, when the new position of a particle dominates its personal best experience, it certainly becomes the personal best. However, if the new position and its personal best are non-dominated, the issue to face is whether to update to the new value or not. Keeping the current personal best position helps the particle explore the local region deeper, which can lead to higher quality solutions. On the other hand, it is also desirable to move to new position to spread out the non-dominated front. Because each decision has its own advantages, the algorithm will randomly pick one of them to become the personal best.

For the near neighbor best, a fitness distance ratio (FDR) which was originally developed to find the neighbor best are modified to handle multiple objective functions as shown in equation (4.4).

$$FDR_{MO} = \frac{\sum_{k=1}^{K} \%\Delta_k}{|\theta_{id} - \psi_{od}|} \quad for\ all\ d = 1 \dots D, i = 1 \dots L$$

$$\%\Delta_k = \frac{[f_k(\Theta_i) - f_k(\Psi_o)]}{|f_k(\Theta_i)|} \tag{4.4}$$

In equation (4.4), $f_k(.)$ is the k^{th} objective function and θ_{id}, ψ_{od} are the values at dimension d of particle i and its neighbor o and D and L are the dimension of a particle and the number of particles in the swarm respectively (refer to Peram et al., (2003) and Veeramachaneni et al., (2003) for more details about FDR with single objective). In the implementation, a very small value ε should be included in the dominators to handle the cases that a dominator might become zero. The amount of improvement that can be made when a neighbor h is chosen is represented by $\%\Delta_k$. By using equation (4.4), the near neighbor best should be the one that is expected to guide a particle to a position that can achieve the most improvement across all objective functions.

In order to prevent the particle from being too sensitive to every change of the swarm, the local best is only updated when the new local particles dominated the current best one.

4.4.Movement strategies

As mentioned in the previous sections, MO problems require the swarm to store its searching experience as a set of non-dominated solutions instead of a single best one. Then, a very key research question is how can a particle effectively use the knowledge of this Elite group to guide it to a better position? Because the target is to identify the near optimal Pareto front, the definition of a better position is more complex than that for the

cases of single objective optimization problems. In literature, the three common criteria to measure the quality of a non-dominated front $\mathcal{N}$ are:

- The average distance to the Pareto optimal front $\mathcal{P}$
- The distribution of non-dominated solutions in $\mathcal{N}$
- The spread of $\mathcal{N}$ in the multi-objective space

Similar to any optimization problem, the gap between the solutions found and the true optimal solutions should be as small as possible. Moreover, the solutions should provide a good outline of the Pareto front so that the decision makers can make more informed decisions. Based on the above criteria, six movement strategies are proposed. These strategies are especially designed to obtain high quality Pareto front. The procedures to perform these movements will be included in step *vii* and *viii* of the MOPSO framework.

4.4.1. Ms1: Pick a global guidance located in the least crowded areas

This strategy aims at diversifying particles in the swarm so that they can put more effort in exploring the less crowded areas, thereby increasing the spread of the non-dominated front. For that reason, a particle in the Elite group with fewer particles surrounding it is preferred when selecting the global guidance.

The crowded distance CD estimates the density of solutions located around a specific solution by calculating the average distance of two points on either side of this point along each of the objectives. A procedure to calculate the crowding distance (CD) for each member in the Elite group is implemented as given in NSGA II. To make this paper self-contained, the algorithm to calculate CDs is given in algorithm CD below.

Algorithm CD: $Calculate\_crowding\_distance\ (\mathcal{E})$ (from Deb et al., 2002)

$L = |\mathcal{E}|$
$For\ each\ i, set\ \mathcal{E}[i].distance = 0$
$For\ each\ objective\ m$
    $\mathcal{E} = sort(\mathcal{E}, m)$
    $\mathcal{E}[1].distance = \mathcal{E}[L].distance = \infty$
    $For\ i = 1\ to\ (L-1)$
        $\mathcal{E}[i].distance = \mathcal{E}[i].distance + (\mathcal{E}[i+1].m - \mathcal{E}[i-1].m)/(f_k^{max} - f_k^{min})$

Particles with higher CDs are located in less crowded area and they are considered to be good candidates for global guidance in this movement strategy. $\psi_{g,d}$ and $\theta_{i,d}$ are dimension d of the global guidance g and particle i in the swarm respectively.

The movement direction of Ms1 is shown in Figure 4.3 and the pseudo-code for this movement strategy is presented in algorithm A2. In step *i* of algorithm A2, a procedure to calculate the crowding distance (CD) for each member in the Elite group $\mathcal{E}$ is called.

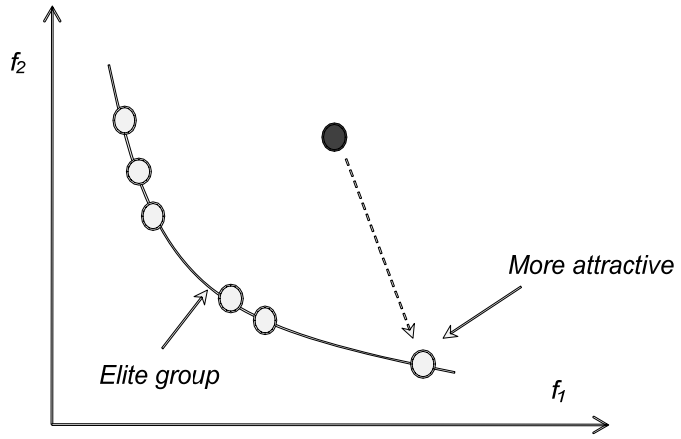

Figure 4.3: Movement strategy 1 in bi-objective space

A2. Algorithm for Ms1

i. *Calculate_crowding_distance* ($\mathcal{E}$)
ii. *Sort* $\mathcal{E}$ *by decreasing order of crowding distance (CD) values*
iii. *Randomly select a particle* g *from top t% of* $\mathcal{E}$
iv. *Update global term in particle i movement by*

$$c_g u\,(\psi_{g,\mathrm{d}} - \theta_{\mathrm{i,d}}) \qquad \textit{for all dimension d with } u \sim U(0,1)$$

4.4.2. Ms2: Create the perturbation with Differential Evolution concept

The fact that more than one global non-dominated solution exist has raised the questions of whether it is better to combine the knowledge of two or more members in the Elite group to guide a particle. In this strategy, the concept of Differential Evolution (DE), proposed by Price and Storn (1995) for continuous function optimization, is adopted to utilize the flying experience of two individual in the Elite group. The key idea behind DE is to use vector differences for perturbing the vector population. In the original DE algorithm, a new parameter vector is generated by adding the weighted difference between two population members to a third member (all of these vectors are randomly selected from the population). A fitness selection scheme similar to Genetic Algorithm (GA) is carried out to produce offspring to form new population.

The inspiration for this strategy is that this PSO has the tendency to converge quite fast to some best solutions in the swarm. This is counterproductive since this can reduce its ability to search for a wider range of solutions in a Pareto front. Therefore, it is more desirable to have a mechanism to perturb the swarm and move its members to the new and less crowded areas. Figure 4 demonstrates the moving strategy Ms2 which adopts the DE concept to create the moving direction for a particle. The algorithm for Ms2 is presented in A3.

The points in Figure 4.4 show the objective values of each particle in objective space; however, it is important to note that that the vectors also represent the corresponding positions of particles as well as their movements in positional space (and these vectors can only be plotted in higher dimension space).

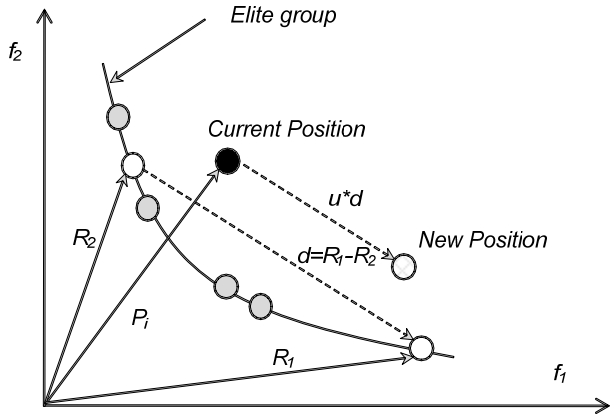

Figure 4.4: Movement strategy 2 in bi-objective space

A3. Algorithm for Ms2

i. *Calculate_crowding_distance* ($\mathcal{E}$)
ii. *Sort* $\mathcal{E}$ *by decreasing order of crowding distance (CD) values*
iii. *Randomly select a particle* R_1 *from top t% of* $\mathcal{E}$
iv. *Randomly select a particle* R_2 *from bottom b% of* $\mathcal{E}$
v. *Update global term in particle i movement by*

$$c_g u\,(\psi_{R1,d} - \psi_{R2,d}) \qquad \text{for all dimension } d \text{ with } u \sim U(0,1)$$

4.4.3. Ms3: Explore the unexplored space in the non-dominated front

The two strategies discussed above focus mainly on moving particles to less crowded areas and expand the spread of the non-dominated front. Here, strategy Ms3 is aimed at filling the gap in the non-dominated front and hence improving the distribution of the solutions in the front. Figure 4.5 shows how the information in the Elite group is used to guide a particle to potential unexplored space within the current non-dominated front.

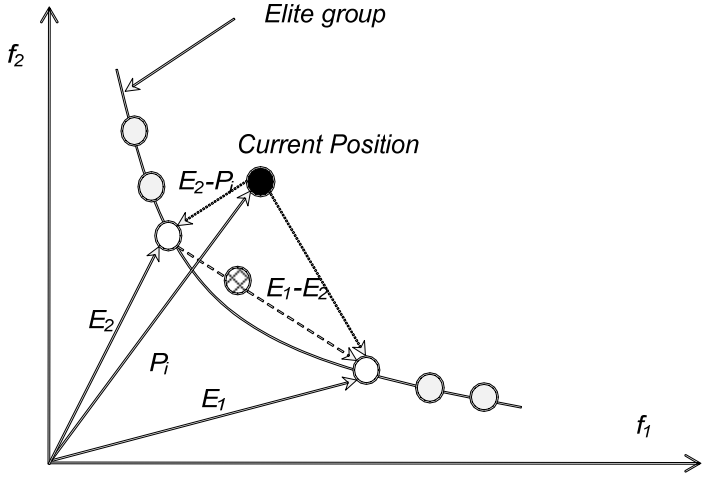

Figure 4.5: Movement strategy 3 in bi-objective space

In this strategy, the first step is to identify the potential gap in the Elite group. When the gap is determined, a pair of vectors is used to represent the gap. Algorithm A4 provides the procedure to identify pairs of unexplored vectors and how to move particle based on this information.

A4. Algorithm for Ms3

i. Identify the unexplored areas in $\mathcal{E}$
For each objective functions $f_k(.)$
Sort $\mathcal{E}$ *in increasing order of objective function* $f_k(.)$
For i=1 to $|\mathcal{E}|$ *-1*
$Gap = f_k(\Theta_{i+1}) - f_k(\Theta_i)$
If $Gap > x\% * (f_k^{max} - f_k^{min})$*:*
add pair (i,i+1) in unexplored list $\mathcal{U}$

ii. Randomly select one pair (E1, E2) from $\mathcal{U}$

iii. Update global term in particle i movement by

$c_g\, u\, [(E_{1,d} - \theta_{i,d}) + r * (E_{1,d} - E_{2,d})]$ *for all dimension d with u,* $r \sim U(0,1)$

The range of objective function $f_k(.)$ in the Elite group is $(f_k^{max} - f_k^{min})$. By using the condition $Gap > x\% * (f_k^{max} - f_k^{min})$, it is expected that the final non-dominated front will only include the gap less than $x\%$ of the any objective function range. Reducing the value of x can improve the distribution of the final front but, at the same time, it may distribute the effort of swarm across the front and slow down the process of searching for better solutions.

4.4.4. Ms4: Combination of Ms1 and Ms2

This strategy tries to balance between the exploration and exploitation abilities of Ms2. Therefore, instead of moving purely to new areas by DE concept, a component similar to Ms1 is added to the perturbation formula in A3 so that a particle not only explores the new region but also benefits from the flying experience of the Elite group to improve the solution quality. Ms4 uses the same algorithm as Ms2 with the following updating formula:

$$c_g u \left[\left(\psi_{R1,d} - \theta_{i,d}\right) + (\psi_{R1,d} - \psi_{R2,d})\right] = c_g u \left(2\psi_{R1,d} - \theta_{i,d} - \psi_{R2,d}\right)$$

4.4.5. Ms5: Explore solution space with mixed particles

Since each of the movement strategies has its own advantages which can have different contributions toward a high quality Pareto front, it would be beneficial to include more than one search strategy in the algorithm. One of the straightforward ways to perform this idea is to use a heterogeneous swarm, i.e., a single swarm with a mixture of particles with different movement strategies. It is preferable that the composition of a productive swarm should include groups of particles with the following characteristics:

- Ones that prefer to explore based on its own experience and with some influence from its neighbors – Group 1

- Ones that prefer to follow the global trend but avoid the crowded areas (Ms1) – Group 2
- Ones that like to explore new areas (Ms2) – Group 3
- Ones that fill the gaps left by previous movements (Ms3) – Group 4

In Ms5, these four groups of particles co-exist in the same swarm and all of their flying experience is stored in a common Elite archive. A particle of the first group will not directly use the global knowledge but will explore the space gradually based on its own experience and a partial knowledge of its neighbor. For that reason, these particles do not change their movement abruptly every time the global trend changed. This feature helps them to better explore the local region. The second group, on the other hand, searches by using the status of the Elite group and moves to the position that has not been well explored. In the cases that particles in the Elite group have distributed uniformly, members in this group will have similar movement behavior as those in the first group. The responsibility of particles in group 3 is to explore the border to increase the spread of the non-dominated fronts with their perturbation ability. Although the first three groups have tried to explore the search in many different directions, they may still leave some gaps unexplored because of their convergence at some segments on the Pareto front. The task of the last group is to move to fill these gaps so that the final front can have a better distribution.

4.4.6. Ms6: Adaptive Weight Approach

The sixth movement strategy Ms6 is the only one that does not use the global Elite group. The swarm follow Ms6 is divided into n + 1 sub-swarms with n is the number objective functions. The first n sub-swarms will search for the optimal solution corresponding to each objective functions just like the tradition PSO. The last sub-swarm will minimize the adaptive weighted function as defined in Gen et al. (2008) by the following formula:

$$F(x) = \sum_{j=1}^{n} w_k(f_k(x) - f_k^{min}) \qquad where\ \ w_k = \frac{1}{f_k^{max} - f_k^{min}} \tag{4.5}$$

4.5.M3PSO library

It is noted that the traditional PSO algorithm needs to be changed to deal with MO problems. Therefore, a new library called M3PSO (Multi-strategy Multi-Learning-Term Multi-Objective Particle Swarm Optimization) is developed based on the original framework of GLNPSO and includes the suggested modifications proposed in previous sections as shown in Figure 4.2 and Algorithm A1. Basically, besides the available routines in GLNPSO, some additional classes and routines are created to deal with multi-objective problems. The new and modified components are listed below:

Class	Name	Type	Description
Particle	NoObj	Integer	number of objective functions to be minimized
	Objective	Array of real number	the objective values or the fitness of the particle
	ObjectiveP	Array of real number	the objective values corresponding to BestP
	crowdDistance	Real	the crowding distance value which is used to indicate the crowdedness of the current position of the particle
	Trap	Array of Integer number	the indicator of how many iteration in which the value at a specific dimension stays unchanged
	type	Integer	the type of a particle (for movement strategy 5 and 6 as introduced in section 4.4)
Swarm	posBest	Array of Integer number	the index (or location) of global best member in the swarm (pParticle[postBest[k]] refers to the particle which found the position resulting in the best objective value of objective function k)
	MinObj/MaxObj	Array of Real number	the minimal and maximal objective value found by the swarm through searching process
	AvgObj	Array of Real number	the average objective values across all particles in the swarm
	movingStrategy	Integer	the index of movement strategy used by the swarm to explore the Pareto front
	particleMix	2D array of real number	particleMix[i,0] and particleMix[i,1] is the accumulative probabilities which are used indicate which particles in the swarm use movement strategy i
	constr	Bool	Indicator of whether the MO problem have constraints or not
	`public void setMovingStrate`	Method	Set the movement strategy of the

	gy(int mS)		swarm
	public void setParticleMix(ArrayList pMix)	Method	Set the particle mix
	public void setConstraintMode(bool ctr)	Method	Set the value of constr
	private static void AssignUnexploreP(Random rnd, ArrayList USpace, ref Particle E1, ref Particle E2)	Method	Select a pair of particles used to indicate the direction to unexplored areas as described in movement strategy 3
	private static void AssignGlobalP(Random rnd, ArrayList Elist, ref Particle P, ref Particle S, double topP, double topS)	Method	Select a particle located in less crowded area (P) and crowded areas (S) with the probability top and tops as described in movement strategy 1 and 2
	private double FDR_Calculate(int n_temp, double FDRBest, int i, int j)	Method	Calculate the modified FDR index
	public void UpdateBest(int nbSize, Random rnd, bool activeNeighbor)	Method	Update learning terms for movement strategy 1 to 5
	public void UpdateBestSingle(int nbSize)	Method	Update learning terms for movement strategy 6
PSO	nObj	Integer	The number of objective functions to be minimized
	moveS	Integer	The movement strategy used by the swarm
	ElististP	Array List	The list of Elite solutions found through the search
	UnExploreSpace	Array List	The list of pairs of particles which used to indicate the direction to unexplored areas as described in movement

			strategy 3
	MaxElististMember	Integer	Upper limit of the `ElististP`
	parmix	Array List	The proportion of members in the swarm assigned to follow each movement strategies
	Constraint	Bool	Indicator of whether the MO problem have constraints or not
	TopEPerc	Real	The percentage of members on the top of the Elite group (in less crowded areas) which can be randomly picked to become the global guidance in movement strategy 1 and 2
	BotEPerc	Real	The percentage of members at the bottom of the Elite group (in crowded areas) which can be randomly picked to become the global guidance in movement strategy 2
	GapPerc	Real	Percentage of the range (corresponding to each objective function) to identify the value which is used as a threshold to determine the gap in movement strategy 3.
	`public void RecruitElite(ArrayList E)`	Method	Recruit elite member from elite group E
	`void updateElististGroup(ArrayList Front)`	Method	Update the elite group to sort out the dominated solutions
	`public void SortEliteP(int nf, bool constr)`	Method	Perform non-dominated sorting procedure on the elite group
	`void crowding_Distance_assignment(ArrayList ElististP)`	Method	Call the `Crowding_Distance_Calculate_perObj` procedure for each objective function
	`private void Crowding_Distance_Calculate_perObj(ArrayList` Method	Method	Calculate the crowding distance corresponding to each objective function

	ElististP, int o)		
	public virtual double[] Objective(Particle p)	Method	Evaluate the all objective values of a particle

4.6.A simple example of multi-objective optimization problem

In this section, M3PSO is applied to solve a simple MO problem. For the ease of illustration, this problem deals with two objective functions but it can be easily modified to handle more than two objective functions. The problem below is the SCH problem which is normally used to test the effectiveness of MO algorithm.

$$Minimize \quad f_1(x) = x^2$$
$$f_2(x) = (x-2)^2$$
$$Where\ x \in [-10^3, 10^3]$$

Similar to single objective optimization discussed in chapter 2, we have to determine the dimension of a particle, the method to evaluate the objective values, and the method to initialize the swarm. In general, M3PSO are designed so that problems can be easily formulated without worrying too much about the optimization algorithms. Figure 4.6 shows how a new class is created to solve the problem with M3PSO.

```
class spPSO : M3PSO
{
   public spPSO(int nPar, int nIter, int nNB, double dwmax, double dwmin,
   double dcp, double dcg, double dcl, double dcn, int maxE, int moveStr,
ArrayList pm,double te,double be, double gap)
   :base(nIter, nNB, dwmax, dwmin, dcp, dcg, dcl, dcn,maxE,moveStr,pm)
   {
     //define problem
     int dimension=1;          //dimension of a particle is 1
     bool constr = false;      //there is no constraint
     int nObj=2;               //two objective functions to be minimized
     if (moveStr==6)
        base.SetParameters(nPar,dimension,nObj+1,constr,te,be,gap);
     else
        base.SetParameters(nPar, dimension, nObj, constr, te, be, gap);
     //number of particles, dimension,
     //number of objective (+1 if ms6 is used, and +1 more if there are
constraints in the model
     //and constraint activator (true if there are any constrains in the
model
    }
   public override void DisplayResult(TextWriter t)
   {
     t.WriteLine("No. NonDom: " + "\t" + "{0}",ElististP.Count);
     for (int i = 0; i < this.ElististP.Count; i++)
     {
      for (int o = 0; o < ((Particle)this.ElististP[0]).NoObj; o++)
      t.Write(((Particle)this.ElististP[i]).Objective[o].ToString()+"\t");
      t.WriteLine();
     }
     t.WriteLine("");
```

```
      t.WriteLine("Result:");
      t.WriteLine("-------");
    }
    public override double[] Objective(Particle p)
    {
      double[] obj=new double[p.NoObj];
      Function.SCH_Function(p, obj);
      return obj;
    }
    public override void InitSwarm()
    {
      for (int i=0; i<sSwarm.Member; i++)
      {
        for (int j = 0; j < sSwarm.pParticle[i].Dimension; j++)
        {
         sSwarm.pParticle[i].Position[j] = -1000 + 2000 * rand.NextDouble();
         sSwarm.pParticle[i].Velocity[j] = 0;
         sSwarm.pParticle[i].BestP[j] = sSwarm.pParticle[i].Position[j];
         sSwarm.pParticle[i].PosMin[j] = -1000;
         sSwarm.pParticle[i].PosMax[j] =  1000;
        }
        for (int o=0;o<sSwarm.pParticle[i].NoObj;o++)
            sSwarm.pParticle[i].ObjectiveP[o] = 1.7E308;
      }
      sSwarm.posBest=new int[sSwarm.pParticle[0].NoObj];
    }
}
class Function
{
    public static void SCH_Function(Particle p, double[] x)
    {
      double var = p.Position[0];
      x[0] = Math.Pow(var, 2);
      x[1] = Math.Pow(var - 2, 2);
    }
}
```

Figure 4.6: Formulate SCH problem in C#

The formulation of MO problem is very similar to that of single objective optimization problem except for the function evaluation method which returns multiple objective values instead of a single value. The M3PSO's parameters are defined in the main class as presented in Figure 4.7.

```
class MainClass
{
  public static void PSO(int fx,double[] PSOparas, int strategy, bool
aniEnable, out double[] index, out ArrayList Pareto, out ArrayList Ani, out
ArrayList AniS, out ArrayList Average)
  {
    ## Animation declaration ##
    //parameter setting
    int noIter = Convert.ToInt32(PSOparas[0]);
    int noPar = Convert.ToInt32(PSOparas[1]);
    double wMin = PSOparas[2];
    double wMax = PSOparas[3];
    int noNB = Convert.ToInt32(PSOparas[4]);
    double cP = PSOparas[5];
    double cG = PSOparas[6];
    double cL = PSOparas[7];
```

```
    double cN = PSOparas[8];
    int maxE = Convert.ToInt32(PSOparas[9]);
    double TopEp = PSOparas[10] / 100;
    double BotEp = PSOparas[11] / 100;
    double GapUnexplore = PSOparas[12] / 100;
    int moveStrategy = strategy;
    bool multiSwarm = false;
    int rSeed = (int)PSOparas[17];
    int noRep = (int)PSOparas[18];
    // end parameter setting
    if (moveStrategy == 6)
      {
        pMix.Add(0); pMix.Add((double)PSOparas[13] / 100);
        pMix.Add(1); pMix.Add((double)PSOparas[14] / 100);
        pMix.Add(2); pMix.Add((double)PSOparas[15] / 100);
      }
    if (moveStrategy == 5)
      {
        pMix.Add(0); pMix.Add((double)PSOparas[13] / 100);
        pMix.Add(1); pMix.Add((double)PSOparas[14] / 100);
        pMix.Add(2); pMix.Add((double)PSOparas[15] / 100);
        pMix.Add(3); pMix.Add((double)PSOparas[16] / 100);
      }
    // starting time and finish time using DateTime datatype
    DateTime start, finish;
     // elapsed time using TimeSpan datatype
           TimeSpan elapsed;
    ## Write parameter to text ##
    for (int i = 0; i < noRep; i++)
      {
        rSeed++;
        AvgVal[i] = new ArrayList();
        Console.WriteLine("Replication {0}", i + 1);
        tw.WriteLine("Replication {0}", i + 1);
        // get the starting time from CPU clock
        start = DateTime.Now;
    // main program ...
    M3PSO GlobalSwarm = new spPSO(fx,noPar, noIter, noNB, wMax, wMin, cP,
          cG ,cL, cN, maxE, moveStrategy, pMix, TopEp, BotEp, GapUnexplore;
    GlobalSwarm.SetRSeed(rSeed);
    GlobalSwarm.Run(tw, true, aniEnable, AvgVal[i], out sAni, out sAni2);
    // get the finishing time from CPU clock
    finish = DateTime.Now;
    elapsed = finish - start;
    // display the elapsed time in hh:mm:ss.milli
    ## Display output ##
   }
}
```

*** the code in ## ... ## contain the subroutine which can be found in the original code*

Figure 4.7: C# implementation of M3PSO algorithm

The more generalized source code of this example, which includes a convenient interface and a list of test problems, can be found in "\GLNPSO manual\GLNPSO basic\PSO_MutiObjective\". This small application also provides the animation feature to help the user easily observe the movement behavior of the algorithm in bi-objective space as shown in the Figure 4.8.

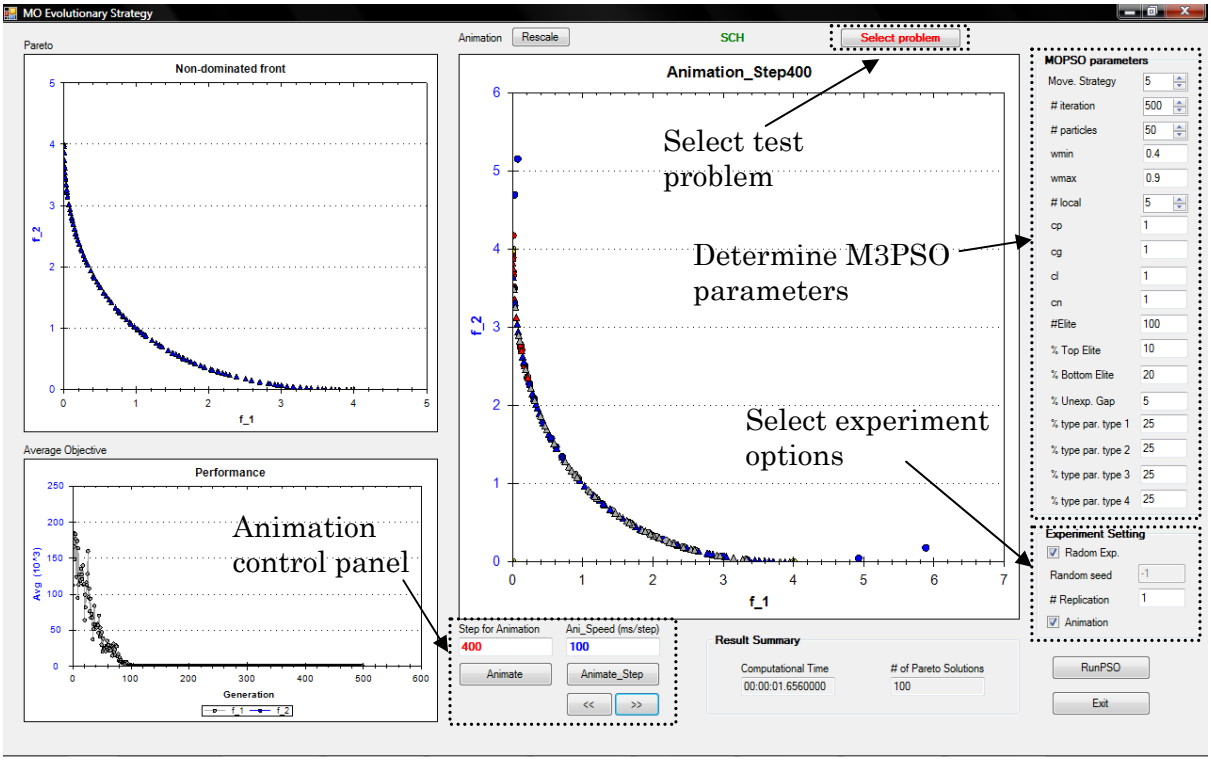

Figure 4.8: Multi-objective optimizer with M3PSO

Figure 4.8 shows the interface built for research purpose. The figure on the upper left corner presents the final Pareto front found by the M3PSO algorithm. The average objective value of each objective function through each step is shown in the figure on the lower left corner. The largest figure in the middle is used for animation. At each animation step, the elite members are represented by triangle point and the current position of each particle is represented by the circle point. The color of each point is used to identify the type of a particle (in movement strategy 5 and 6). Table 4.1 shows how meaning of colors used in animation screen.

Table 4.1: Color set used for animation

Color	Ms1	Ms2	Ms3	Ms4	Ms5	Ms6
Yellow	Type 0	Type 0	Type 0	Type 0	Type 0	Type 0
Gray	Na	Na	Na	Na	Type 1	Type 1
Blue	Na	Na	Na	Na	Type 2	Type 2
Red	Na	Na	Na	Na	Type 3	Na

When movement strategies Ms1-Ms4 are used, all particles only follow single movement behavior so only one color is used. In movement strategy Ms5, type 0, 1, 2, 3 indicate the particle in group 1, 2, 3, 4 respectively. Meanwhile, type 0 and type 1 in Ms6 represent the particles in the sub-swarms that minimize single objective function 1 and 2 respectively. In Ms6, type 2 indicates the particles in the sub-swarm assigned to minimized adaptive weighted function.

4.7.Portfolio optimization with M3PSO algorithm

Portfolio Optimization (PO) is a critical problem in finance in order to find an optimal way to distribute a given budget on a set of available assets. Although many investment decisions are normally made on qualitative basis, there are an increasing number of quantitative approaches adopted.

The most seminal mathematical model was initiated by Markowitz more than 50 years ago and there have been many extensions of his models since then. The classical mean-variance portfolio selection problem of proposed by Markowitz can be given as:

$$Minimizing\ the\ variance\ of\ the\ portfolio \sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j \sigma_{ij}$$

$$Maximizing\ the\ expected\ return\ of\ the\ portfolio \sum_{i=1}^{N} w_i \mu_i$$

$subject\ to$:

$$\sum_{i=1}^{N} w_i = 1$$

$$0 \le w_i \le 1\ for\ \forall i = 1 \dots N$$

The basic assumption in this model is that asset returns follow multivariate normal distribution. The decision variable w_i is the proportion of the budget which is distributed to asset i. Parameter μ_i and σ_{ij} are the expected return of asset i and the covariance between asset i and j. Because it is difficult to weigh the two criteria before the alternatives are known, the popular approach in this case is to search for the whole efficient frontier. In this section, we will use M3PSO library to solve the portfolio optimization problem.

In this problem the decision variable w_i can be modeled as the particle position which ranging from 0 to 1. However, because the sum of all values of w_i must be equal to 1, positions of particles cannot guarantee to provide feasible solutions. Fortunately, an infeasible solution can be easily repaired to become a feasible one. To illustrate the encoding/decoding scheme, we use a simple example with 4 assets. The data for this problem is provided in Table 4.2. The encoding/decoding scheme for the portfolio optimization problem is shown in Figure 4.9.

Table 4.2: Four asset example

Asset	Expected Return	Std. Deviation	Corelation Matrix			
			1	2	3	4
1	0.004798	0.046351	1	0.118368	0.143822	0.252213
2	0.000659	0.030586		1	0.164589	0.099763
3	0.003174	0.030474			1	0.083122
4	0.001377	0.035770				1

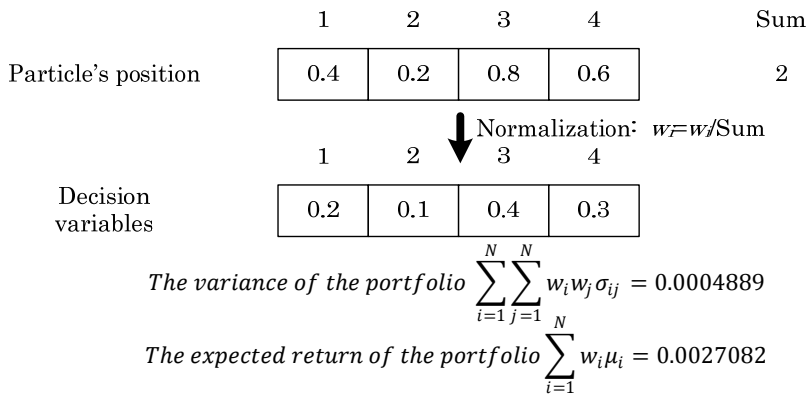

Figure 4.9: Encoding/decoding scheme for classical portfolio optimization problem

Similar to the TSP problem in chapter 3, we built a separate class to get the input data, pre-calculate the covariance matrix and calculate the objective values based on positions of particles at each iteration. The source code and the test problems can be found at "\GLNPSO manual\GLNPSO basic\PSO_MutiObjective-Portfolio Optimization\". The defaulted name of the input file is "Example.txt" and the format of this file is give as:

```
number of assets (N)
for each asset i (i=1,...,N):
  mean return, standard deviation of return
for all possible pairs of assets:
  i, j, correlation between asset i and asset j
```

Figure 4.10 shows the application to solve portfolio optimization problem based on M3PSO library.

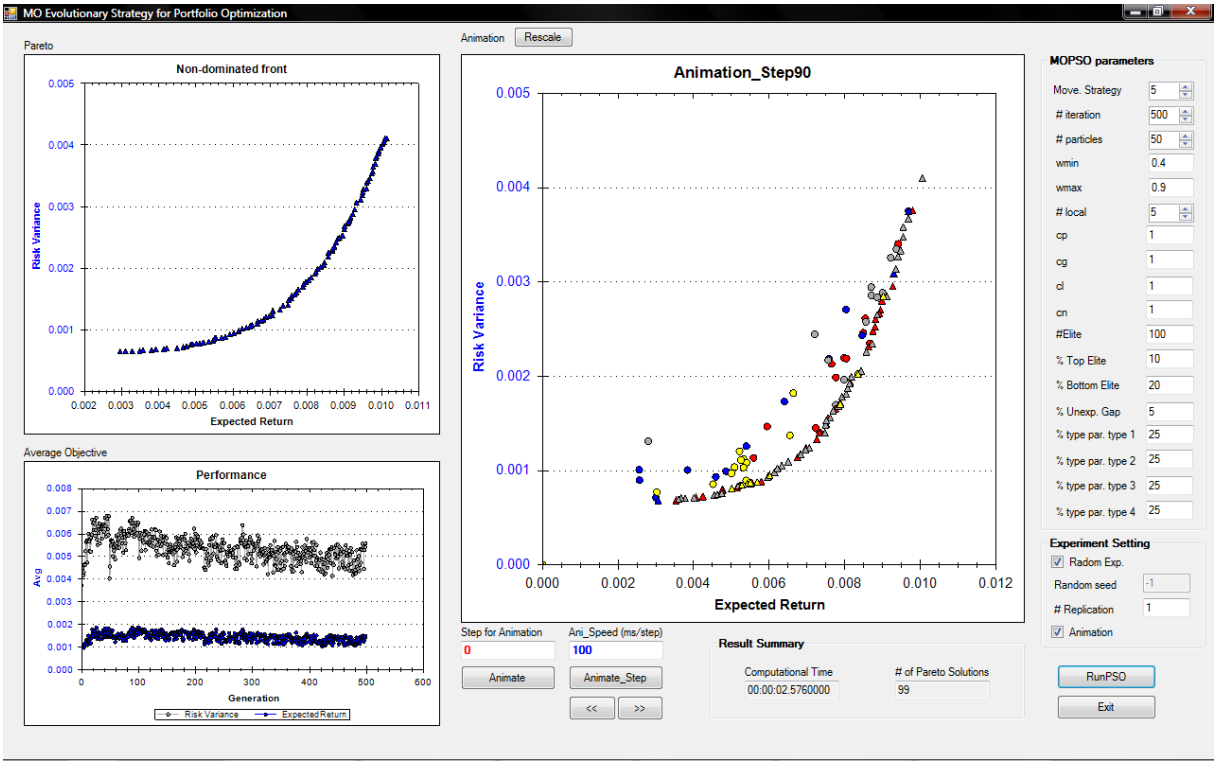

Figure 4.10: Portfolio optimizer with M3PSO library

4.8.Multi-objective optimization in Engineering Design

Our objective is to find the dimension of an I-beam as shown in Figure 4.11, which have to satisfy the geometric and strength constraints and minimize following objective functions:

- Cross section area of beam
- Static deflection of the beam under a certain force

The mathematical model of this problem by Coello and Christiansen[1] are given as follows:

[1] Coello and Christiansen (1999), MOSES: a multiple objective optimization tool for engineering design. J Eng Optim 1999; 31(3):337–68.

$$f_1(\vec{x}) = 2x_2x_4 + x_3(x_1 - 2x_4) \quad (cm)$$

$$f_2(\vec{x}) = \frac{60000}{x_3(x_1 - 2x_4)^3 + 2x_2x_4[4x_4^2 + 3x_1(x_1 - 2x_4)]}$$

Subject to:

$$g(\vec{x}) = 16 - \frac{180000x_1}{x_3(x_1 - 2x_4)^3 + 2x_2x_4[4x_4^2 + 3x_1(x_1 - 2x_4)]} - \frac{15000x_2}{(x_1 - 2x_4)^3x_3^3 + 2x_4x_2^3} \geq 0$$

$$10 \leq x_1 \leq 80, \quad 10 \leq x_2 \leq 50, \quad 0.9 \leq x_3 \leq 5, \quad 0.9 \leq x_4 \leq 5$$

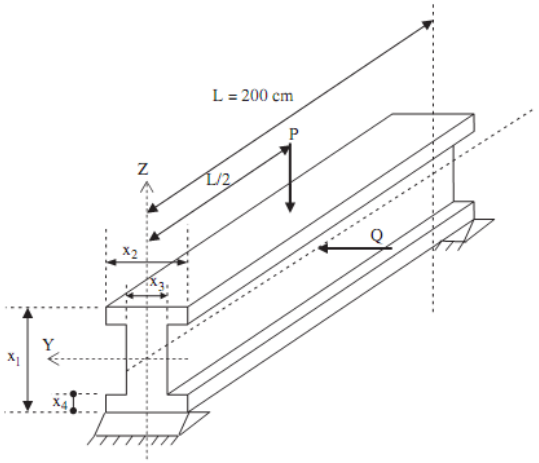

Figure 4.11: I-Beam design problem

Since the objective functions of this problem are very well-defined, we can directly use the values of particle's position as those of decision vector $\vec{x}$. Therefore, the dimension of particle needs is 4 and each dimension will have the upper and lower bounds corresponding to those defined in the mathematical model. The implementation of this problem can be found in the group of test problems in section 4.6. The illustrative example of the Pareto solution for this problem is given Figure 4.12.

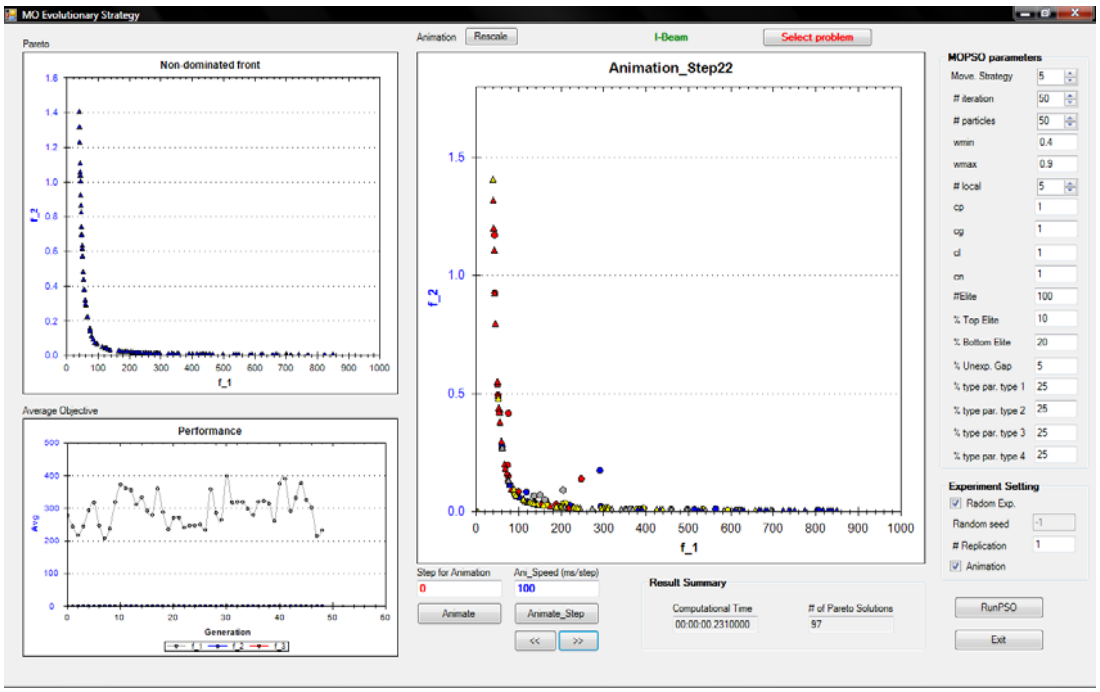

Figure 4.12: Solve I-Beam design problem with M3PSO

Bibliography Of Works Utilizing ET-Lib

1. Kasemset, C. and Kachitvichyanukul, V.
   Bi-level multi-objective mathematical model for job-shop scheduling: the application of Theory of Constraints, International Journal of Production Research, DOI: 10.1080/00207540903176705, November 2009.

2. Ai, The Jin, and Kachitvichyanukul, V.
   A Particle Swarm Optimization for Vehicle Routing Problem with Time Windows, International Journal of Operational Research, Vol. 6, No. 4, pp519-537, 2009

3. Ai, The Jin, and Kachitvichyanukul, V.
   A Particle Swarm Optimization for the Heterogeneous Fleet Vehicle Routing Problem, International Journal of Logistics and SCM Systems, Vol. 3, No. 1, pp32-39, 2009

4. Ai, The Jin, and Kachitvichyanukul, V.
   A particle swarm optimization for the vehicle routing problem with simultaneous pickup and delivery, Computers & Operations Research, 36, pp1693-1702, 2009.

5. Ai, The Jin, and Kachitvichyanukul, V.
   Particle Swarm Optimization and Two Solution Representations for Solving the Capacitated Vehicle Routing Problem, Computers & Industrial Engineering, Volume 56, Issue 1, pp380-387, 2009.

6. Ai, The Jin, and Kachitvichyanukul, V.
   A Particle Swarm Optimization for the Capacitated Vehicle Routing Problem, International Journal of Logistic and SCM Systems, Volume 2, Number 1, pp50-55, 2007

7. Kachitvichyanukul, V. and Dao Duc Cuong
   A Mixed Particle Swarm Optimization Algorithm for Continuous-flow-shop Scheduling Problem, the 20th International Conference on Production Research, Shanghai, China, August 2009

8. Ai, The Jin, and Kachitvichyanukul, V.
   A Study on Adaptive Particle Swarm Optimization for Solving Vehicle Routing Problems, Proceedings of the 9th Asia Pacific Industrial Engineering and Management Systems Conference (APIEMS 2008), Bali, Indonesia, December 2008.

9. Ai, The Jin, and Kachitvichyanukul, V.
   Adaptive Particle Swarm Optimization Algorithms, Proceedings of the 4th International Conference on Intelligent Logistics Systems（ILS2008）, Shanghai, China August 2008

10. Pratchayaborirak, T., and Kachitvichyanukul, V.
    A Comparison of GA and PSO Algorithm for Multi-objective Job Shop Scheduling Problem, Proceedings of the 4th International Conference on Intelligent Logistics Systems（ILS2008）, Shanghai, China August 2008

11. Ai, The Jin, and Kachitvichyanukul, V.
Dispersion and Velocity Indices for Observing Dynamic Behavior of Particle Swarm Optimization, IEEE Congress on Evolutionary Computation, Singapore, September 2007

12. Ai, The Jin, and Kachitvichyanukul, V.
A Particle Swarm Optimization for the Vehicle Routing Problem with Clustered Customers, Proceedings of the APIEMS 2007 Conference, Taiwan, December 2007

13. Pratchayaborirak, T., and Kachitvichyanukul, V.
A Two-Stage Particle Swarm Optimization for Multi-Objective Job Shop Scheduling Problems, Proceedings of the APIEMS 2007 Conference, Taiwan, December 2007