ETLib User's Guide Volume: 2 Differential Evolution

ASIAN INSTITUTE OF TECHNOLOGY

AIT

ASIAN INSTITUTE OF TECHNOLOGY

ET-Lib User's Guide Volume 2 Differential Evolution

Volume 2.0

July 2013

Su Nguyen

Voratas Kachitvichyanukul

Warisa Wisittipanich

Industrial & Manufacturing Engineering

School of Engineering & Technology

Asian Institute of Technology

THAILAND

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION TO DIFFERENTIAL EVOLUTION ALGORITHM

1.1 Overview

Differential Evolution (DE) is first proposed by Storn and Price in 1995 for global optimization over continuous search space. DE is one of the latest Evolutionary Algorithms (EA) which has increasingly gained attention from researchers for solving many optimization problems.

DE is a population-based random search method where an initial population of size N of D-dimensional vectors is randomly generated, and a new population is generated through the cycles of calculations. A solution in DE algorithm is represented by D-dimensional vector, and each value in the D-dimensional space is represented as a real number. The key idea behind DE which makes the algorithm different from other EAs is its mechanisms for generating new solutions, called trial vectors, by mutation and crossover operation. In DE, each vector is served as a target vector which is then combined with other vectors in the population to form a new vector, called a mutant vector. Next, the mutant vector is crossover with its corresponding target vector to obtain a trial vector. Then, the selection or replacement of an individual occurs only if the trial vector outperforms its corresponding target vector. The evolution process of DE population continues through repeated cycles of three main operators; mutation, crossover, and selection until some stopping criteria are met.

The theoretical framework of DE is relatively simple and inexpensive in terms of computational time. Due to its advantages of relatively few control variables but performing well in convergence and relatively easy to implement, DE has become widely applied and shown its strengths in many application areas from scientific, engineering, to financial such as curve fitting, artificial neural networks training, signal processing, chemistry industry, and production scheduling (Quan et al., 2007, Qian et al., 2008, Godfrey et al., 2006).

The theoretical framework of the classical DE and its variants will be described in the following sections.

1.2 The Classic Form of DE

The classic version of DE is the simplest and most popular scheme found in literatures. The mutation operator in the classic DE generates a mutant vector by adding a weighted difference between two randomly selected vectors to the third randomly selected vector. This scheme has demonstrated to be efficient in solving several multimodal optimization problems (Price et al., 2005 and Chakraborty, 2008). The notations used in the DE algorithm are listed below.

1.2.1 Notations

Indices:

i : Vector index within population of size N, $i = 1, 2, 3, \ldots, N$

j : Dimension index, $j = 1, 2, \ldots, D$

g : Generation index, $g = 1, 2, 3, \ldots, G$

Parameters:

u : Uniform random number in the range [0,1]

j_u : A random chosen index, $j_u \in \{1, 2, 3, \ldots, D\}$

$\check{s}$: A starting position index in exponential crossover scheme, $\check{s} \in \{1, 2, 3, \ldots, D\}$

L : A number of dimension lengths in exponential crossover scheme, $L \in \{1, 2, 3, \ldots, n\}$

F : Scale factor

C_r : Crossover rate, $C_r \in [0,1]$

b_U : Upper bound value of vector position

b_L : Lower bound value of vector position

Variables:

X^g : A set of target vectors at g^{th} generation, $X^g=[\vec{X}_1^g, \vec{X}_2^g, \ldots, \vec{X}_N^g,]$

$\vec{X}_i^g$: The i^{th} target vector at g^{th} generation, $\vec{X}_i^g =[x_{i,1}^g, x_{i,2}^g, \ldots, x_{i,D}^g]$

$\vec{V}_i^g$: The i^{th} mutant vector at g^{th} generation, $\vec{V}_i^g=[v_{i,1}^g, v_{i,2}^g, \ldots, v_{i,D}^g]$

$\vec{Z}_i^g$: The i^{th} trial vector at g^{th} generation, $\vec{Z}_i^g = [z_{i,1}^g, z_{i,2}^g, \ldots, z_{i,D}^g]$

$\vec{X}_{best}^t$: The global best vector at g^{th} generation, $\vec{X}_{best}^g= [x_{best,1}^g, x_{best,2}^g, \ldots, x_{best,n}^g]$

$x_{i,j}^g$: The i^{th} target vector at j^{th} dimension at g^{th} generation

$v_{i,j}^g$: The i^{th} mutant vector at j^{th} dimension at g^{th} generation

$z_{i,j}^g$: The i^{th} trial vector at j^{th} dimension at g^{th} generation

$x_{best,j}^g$: The global best vectors at j^{th} dimension at g^{th} generation

$f(\vec{X}_i^g)$: Fitness value of $\vec{X}_i^g$

$f(\vec{Z}_i^g)$: Fitness value of $\vec{Z}_i^g$

1.2.2 The classic DE framework

The framework of classic DE is illustrated in the Figure 1.1.

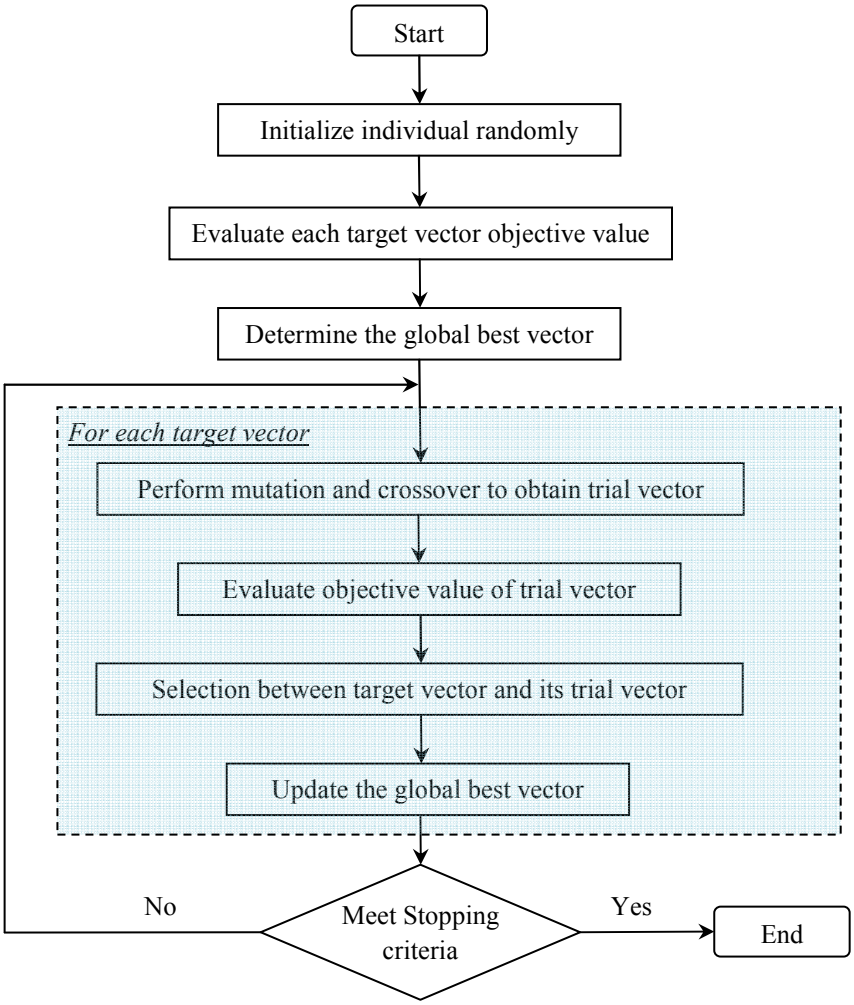

Figure 1.1 Evolution procedures of the classic DE

The detailed procedures of the classic DE framework (Price et al., 2005) are described as follows.

1.2.2.1 Initialize population

DE starts with initializing the population of size *N* of *D*-dimensional vectors, so called target vectors, $\vec{X}_i^g$. The lower bound, b_L, and upper bound, b_U, for the value in each dimension j^{th} (j =1,2,…, *D*) must be specified. At initialization step ($g = 0$), the j^{th} value of the i^{th} vector is randomly generated as shown in equation (1.1) where u is a uniformly random in the interval (0, 1).

$$x_{i,j}^0 = u * (b_U - b_L) + b_L \quad (1.1)$$

1.2.2.2 Evaluating the fitness value of all vectors

For a given target vector, $\vec{X}_i^g$, i =1, 2, 3, …, *N*, it fitness value, $f(\vec{X}_i^g)$, is evaluated.

1.2.2.3 Determining the global best vector

For i =1, 2, 3, …, N, the global best vector, $\vec{X}_{best}^{g}$, which has the best fitness value is determined.

1.2.2.4 Perform mutation

DE performs mutation operation by combining randomly selected vectors to produce mutant vectors. In the classic DE, for a given target vector, $\vec{X}_i^g$, at generation g^{th}, the mutant vector, $\vec{V}_i^g$, is generated according to the following equation.

$$\vec{V}_i^g = \vec{X}_{r1}^g + F * (\vec{X}_{r2}^g - \vec{X}_{r3}^g) \tag{1.2}$$

It is noted that $\vec{X}_{r1}^g, \vec{X}_{r2}^g$, and $\vec{X}_{r3}^g$ are vectors randomly selected from the current population. They are mutually exclusive and different from the target vector, $\vec{X}_i^g$. F is a scale factor which controls the scale of the difference vector between $\vec{X}_{r2}^g$ and $\vec{X}_{r3}^g$. Figure 1.2 graphically illustrates the two-dimensional example for generating a mutant vector, $\vec{V}_i^g$, in the classic DE scheme.

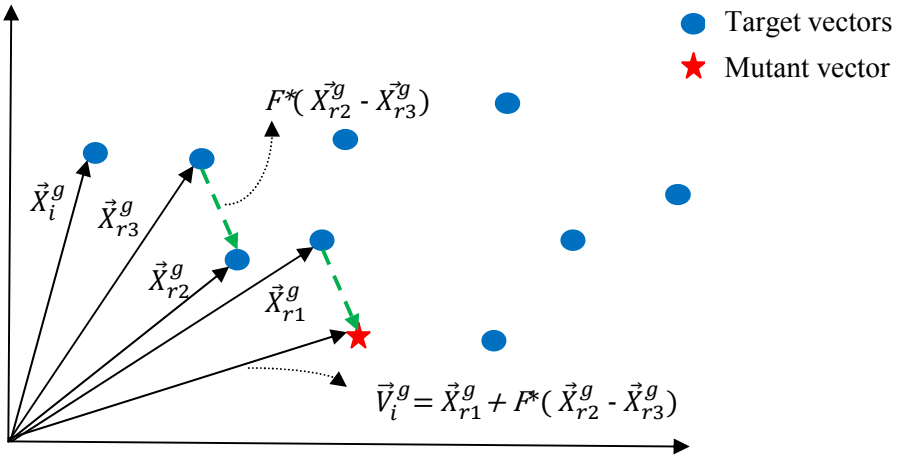

Figure 1.2 Two-dimensional example showing the process for generating a mutant vector in the classic DE

1.2.2.5 Perform crossover

DE applies a crossover operator on $\vec{X}_i^g$ and $\vec{V}_i^g$ in order to obtain the trial vector, $\vec{Z}_i^g$. Generally, two types of crossover operators commonly used in DE are binomial crossover (bin) and exponential crossover (exp).

The binomial crossover is operated on each j^{th} dimension of $\vec{X}_i^g$ and $\vec{V}_i^g$, and the trial vector, $\vec{Z}_i^g$ is generated by equation (1.3).

$$z_{i,j}^g = \begin{cases} v_{i,j}^g, & \text{if } u_j \leq C_r \text{ or } j = j_u \\ x_{i,j}^g, & \text{otherwise} \end{cases} \tag{1.3}$$

Where

u_j : a uniformly random number between [0, 1] for the j^{th} dimension

The exponential crossover scheme for generating a trial vector is expressed in equation (1.4).

$$z_{i,j}^{g} = \begin{cases} v_{i,j}^{g} \text{ , if } j = <\check{s}>_D, <\check{s}+1>_D, \ldots, <\check{s}+L\text{-}1>_D \\ x_{i,j}^{g} \text{ , otherwise} \end{cases} \quad (1.4)$$

The angular brackets $< >_D$ express a modulo function with modulus D (Storn and Price, 1995). Starting at the randomly picked dimension index $\check{s}$, each dimension value of trial vector are inherited from its mutant vector, $\vec{V}_i^g$, as long as $u_j \leq C_r$. The first time when $u_j > C_r$, the value of all the remaining dimensions are taken from the target vector, $\vec{X}_i^g$. An integer, L, indicates the number of consecutive dimension indexes on which crossover is performed.

It is important to note that the crossover probability C_r controls the selection of the value in each dimension from a mutant vector over its corresponding target vector. Figures 1.3 and 1.4 illustrate how a trial vector is generated from binomial crossover and exponential crossover, correspondingly.

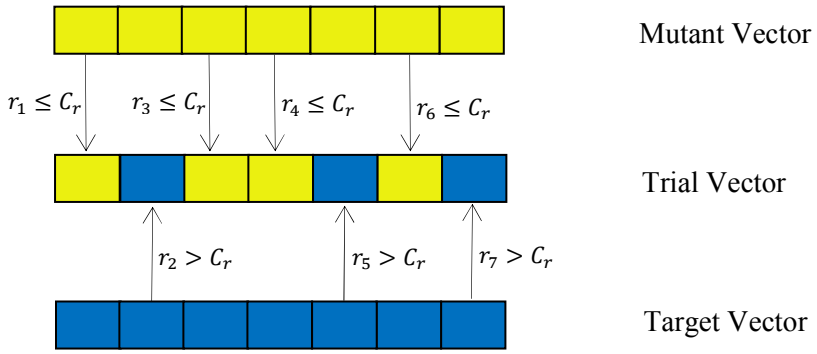

Figure 1.3 Illustration of a mutant vector generated by binomial crossover (bin)

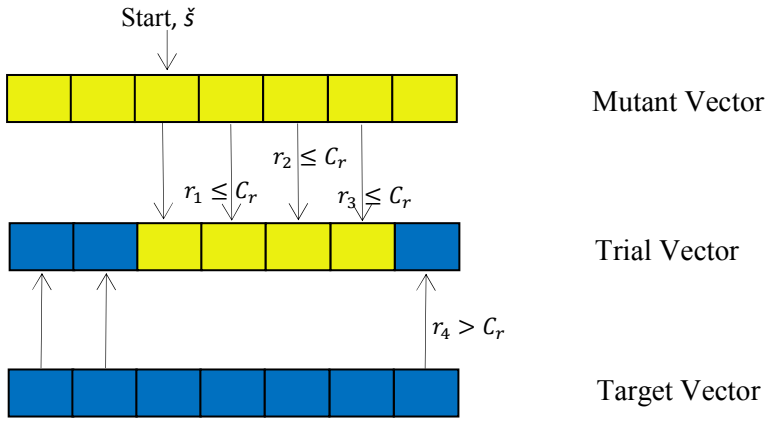

Figure 1.4 Illustration of a mutant vector generated by exponential crossover (exp)

1.2.2.6 Evaluate the fitness value of trial vectors

For each trial vector, $\vec{Z}_i^g$, i =1, 2, 3, …, N, its fitness value $f(\vec{Z}_i^g)$ is evaluated.

1.2.2.7 Selection

The selection operation is performed on each target vector, $\vec{X}_i^g$, and its corresponding trial vector, $\vec{Z}_i^g$, to determine the survival vector for the next generation. The vector, $\vec{X}_i^{g+1}$, is selected according to the greedy criteria as follows.

$$\vec{X}_i^{g+1} = \begin{cases} \vec{Z}_i^g & \text{, if } f\left(\vec{Z}_i^g\right) \leq f(\vec{X}_i^g) \\ \vec{X}_i^g & \text{, otherwise} \end{cases} \tag{1.5}$$

It should be noted that this selection scheme is applicable for minimization problem.

1.2.2.8 Update the global best vector

For, i =1, 2, 3, …, N, the global best vector, $\vec{X}_{best}^{g+1}$, with the best fittest value is determined and updated.

1.2.2.9 Selection

If a stopping criterion is met, i.e $g = G$, stop, and the output is the global best vector and its fitness value. Otherwise $g = g + 1$ and return to step 1.2.2.4.

Following these procedures, all individuals or solutions vectors in the next generation are as good as or better than their counterparts in the current generation.

The general notation used to represent DE variants is DE$/a/b/c$, where

DE stands for DE

a represents the type of vector to be perturbed which can be "rand" (a randomly chosen vector) or "best" (the best vector with the best fittest value in the population)

b is the number of vectors used for perturbation of c

c denotes the type of crossover being used

The classic DE is denoted as DE/rand/1/bin where binomial crossover is used.

1.3 Complete DE family

Currently, there are several DE variants. These variants are differentiated mainly in the mutation mechanisms. In addition to the classic DE, Price et al. (2005) proposed the other four variants of DE with different mutation schemes as listed below.

DE/current-to-best/1 : $\vec{V}_i^g = \vec{X}_i^g + F_1 * \left(\vec{X}_{best}^g - \vec{X}_i^g\right) + F_2 * \left(\vec{X}_{r1}^g - \vec{X}_{r2}^g\right)$ (1.6)

DE/best/1 : $\vec{V}_i^g = \vec{X}_{best}^g + F * (\vec{X}_{r1}^g - \vec{X}_{r2}^g)$ (1.7)

DE/best/2 : $\vec{V}_i^g = \vec{X}_{best}^g + F_1 * (\vec{X}_{r1}^g - \vec{X}_{r2}^g) + F_2 * (\vec{X}_{r3}^g - \vec{X}_{r4}^g)$ (1.8)

DE/rand/2 : $\vec{V}_i^g = \vec{X}_{r1}^g + F_1 * (\vec{X}_{r2}^g - \vec{X}_{r3}^g) + F_2 * (\vec{X}_{r4}^g - \vec{X}_{r5}^g)$ (1.9)

Where $\vec{X}_{r1}^g$, $\vec{X}_{r2}^g$, $\vec{X}_{r3}^g$, $\vec{X}_{r4}^g$, and $\vec{X}_{r5}^g$ are randomly selected vectors from the population in the current generation. F_1 and F_2 are also scale factors which may be the same to reduce the number of parameters.

Each mutation strategy is combined with either the "binomial (bin)" or the "exponential (exp)" type crossover. As a result, this combination yields the total of 10 DE schemes (Price et al., 2005) as summarized as follows.

1. DE/rand/1/bin
2. DE/rand/1/exp
3. DE/current-to-best/1/bin
4. DE/current-to-best/1/exp
5. DE/best/1/bin
6. DE/best/1/exp
7. DE/best/2/bin
8. DE/best2/exp
9. DE/rand/2/bin
10. DE/rand/2/exp

Even though DE has been proven to be one of the most efficient evolutionary algorithms providing promising results in many optimization problems, it still contains some weakness i.e. premature convergence. To remedy these drawbacks and enhance DE performance, many DE variants have been proposed with the main effort to differently regulate the selection of vector for generating a new solution.

1.4 DE Control Parameters

The three critical control parameters of DE are the population size (*N*), the scale factor (*F*), and crossover rate (C_r). These parameters required a proper setting to ensure the DE performance. Generally, generating larger *N* provided more robustness in the algorithm. However, this results in a high cost of computational time. The crossover rate C_r indicates the difference between the trial vector and its corresponding target vector. A lower value of C_r helps maintaining the diversity of population but may reduce the speed of convergence. In contrast, larger C_r creates a big difference between the target and trial vector which helps the exploitation search of DE and accelerates the convergence rate (Wu et al., 2006). The scale factor (*F*) has a large effect on the convergence rate and population diversity. A large value of *F* enhances global exploration to increase the chance of finding global optima, but it may lead to a reduction of convergence speed. On the other hand, a small value of *F* leads to sufficient exploitation and increases the accuracy of global optimum, but it may also decrease the likelihood of global convergence (Nobakhi and Wang, 2008).

In practices, the choice of DE control parameters is primarily based on empirical analysis and experimental results.

CHAPER 2

STRUCTURE OF DE LIBRARY

Before discussing each component in DE library, we will demonstrate how the program works by a simple example. The source code of the example can be found in "\Basic Models\DE 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, the objective is to minimize an objective function $f(\vec{x}) = \sum_{i=1}^{n}[0.01{x_i}^2 + 2 * \sin(x_i)\,] \, where \; \vec{x} = \{x_1, x_2, \dots, x_n\}, x_i \in [-100,100] \; with \; \forall i$. 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, DE 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 DE algorithm, we start by solving the problem with n=1.

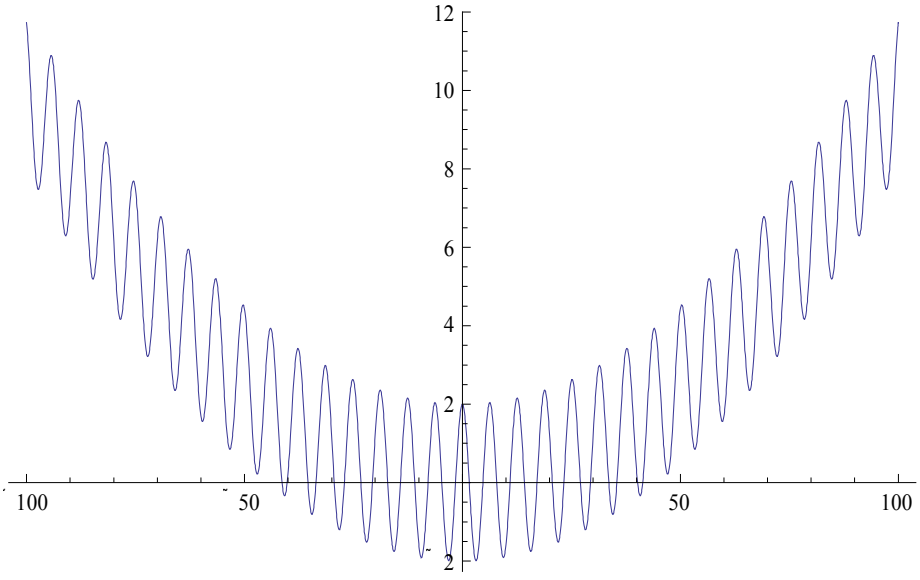

Figure 2.1: Function with multiple local minimum

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

- What is the dimension of a vector?
- How to evaluate the fitness of a vector?
- How can the DE population be initialized?

In case of n=1, the position of a vector is defined as a real number x in the ranges (-100, 100) and the vector's dimension is 1. The objective function $f(\vec{x}) = f(x)$ is used to measure the fitness of each vector (DE Library 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 vector is considered to be 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 population is created by randomly generating the position of each vector in the population, which means that each position will follow the Uniform Distribution with the lower bound of -100 and upper bound of 100.

```
class newDE : DE
{ // this part is the problem specific code
   // Minimize f(x) = 0.001x^2 + 2*Sin(x), -100<=x<=100

    public newDE(int nVec, int nIter, double Fmax, double Fmin, double
croRx, double croRn)
    : base(nIter, Fmax, Fmin, croRx, croRn)
    {
        base.SetDimension(nVec, 1);
    }
    //Define objective function
    public override double Objective(DecisionVector P, int trial)
    {
        double obj = 0;
        if (trial == 0)
        {
            for (int i = 0; i < P.Dimension; i++)
                obj += 0.001 * Math.Pow(P.CurrentVector[i], 2)+ 2*
                               Math.Sin(P.CurrentVector[i]);
        }
        if (trial == 1)
        {
            for (int i = 0; i < P.Dimension; i++)
                obj += 0.001 * Math.Pow(P.TrialVector[i], 2)+ 2*
                               Math.Sin(P.TrialVector[i]);
        }
        return obj;
    }
    //Initialize population
    public override void InitPop()
    {
        for (int i = 0; i < Pop.Member; i++)
        {
            for (int j = 0; j < Pop.Vector[i].Dimension; j++)
            {
                Pop.Vector[i].CurrentVector[j] = -100+200*rand.NextDouble();
                Pop.Vector[i].VecMax[j] = -100;
                Pop.Vector[i].VecMin[j] = 100;
            }
            Pop.Vector[i].Objective = 1.7E308;
        }
        Pop.posBest = 0;
    }

}
```

Dimension

Calculate objective value

Randomize initial positions and set boundary of position for each vector

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

Figure 2.3 shows the main class in which we define the DE parameters and run the DE algorithm with these parameters. In this experiment, *FMax* and *FMin* are the maximum and

minimum values that are set for scale factor, *F*. Similarly, *CRx* and *CRn* are the maximum and minimum values that are set for crossover rate, *Cr*. The search space is explored by a swarm of size 20 in 100 iterations and five replications are performed. The final solutions and some statistics are reported in "MyDE.xls" at the same folder of the execution file (\Basic Models \DE basic\DE basic\bin\Debug).

```
class MainClass
{
        public static void Main(string[] args)
        {
        int noVec = 20;
        int noIter = 100;
        double FMax = 2;
        double FMin = 0.1;
        double CRx = 0.5; //max crossover rate
        double CRn = 0.1; //min crossover rate
        string oFile = "MyDE.xls";
        int noRep= 5;

        // 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 Vector  ", noVec);
        tw.WriteLine("{0} Number of Iteration ", noIter);
        tw.WriteLine("{0} Parameter Fmax      ", FMax);
        tw.WriteLine("{0} Parameter Fmin      ", FMin);
        tw.WriteLine("{0} Parameter CRmax     ", CRx);
        tw.WriteLine("{0} Parameter CRmin     ", CRn);
        tw.WriteLine("{0} Output File Name    ", oFile);
        tw.WriteLine("");

        for(int i=0; i<noRep; i++)
        {
                tw.WriteLine("Replication {0}", i+1)
                // get the starting time from CPU clock
                start = DateTime.Now;

                // main program ...
                DE myDE = new newDE(noVec, noIter,FMax, FMin, CRx, CRn);
                Console.WriteLine("Replication {0}", i + 1);
                myDE.Run(tw, true);
                myDE.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();
        Console.ReadKey();
        }
}
```

Define DE parameters

Create new DE object and pass the DE parameters

Figure 2.3: Main class for single variable problem

2.2 DE Library Components

The class view of DE Library is presented in Figure 2.4. Generally, there are three important classes required for DE which are DecisionVector, Population and DE.

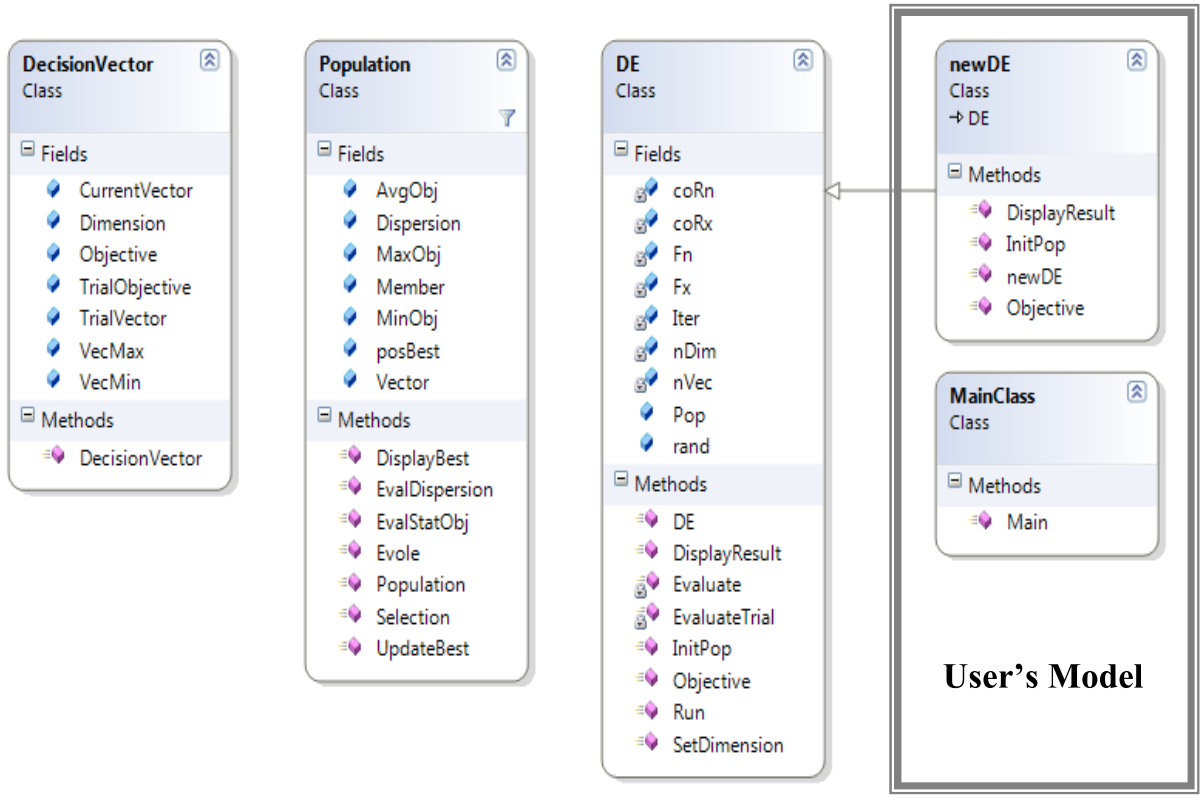

Figure 2.4: Class view for DE in ET-library

2.2.1 DecisionVector class

DecisionVector is the basic class in ET-Lib which includes all information related to a particular vector. The following are the definition of attributes of a vector.

Name	Type	Description
`CurrentVector`	Array of real number	*D*-dimension position of a current vector (or known as a target vector)
`TrialVector`	Array of real number	*D*-dimension position of a trial vector
`Dimension`	Integer	The number of dimensions of vectors
`VecMin/VecMax`	Array of real number	The lower and upper bounds of positions at each dimension
`Objective`	Real	The objective value or the fitness of the current vector

`TrialObjective`	Real	The objective value or the fitness of the trial vector

The constructor `public DecisionVector(int nDim)` is used to create a new vector. The parameter `nDim` indicates the dimension of a vector.

2.2.2 Population class

A population consists of many vectors to explore solutions in the search space. The population class includes all the required activities to regulate the searching behavior of its members (vectors). The attributes and methods of this class are listed below:

Attributes:

Name	Type	Description
`Member`	Integer	Number of vectors in the population (population size)
`Vector`	Array of Vector	A set of vectors in the population
`MinObj/MaxObj`	Real	The minimal and maximal objective value found by the population through the searching process
`AvgObj`	Real	The average objective values across all vectors in the population
`posBest`	Integer	The index of the global best member in the population (`Vector[posBest]` refers to the vector which found the position resulting in the best objective value)
`Dispersion`	Real	The Dispersion index to measure the dispersion of vectors in the population

Methods:

`public Population(int nVec, int nDim)`	Create a new population by determining the number of vectors in the population (`nVec`) and the Dimension of each vector
`public void Evole(int M ,double F,double coRate,Random rnd)`	Create a trial vector by mutation and crossover operator. The parameters passed to this method are the vector index, scale factor, crossover rate, and a random number.
`public void Selection(int M)`	Compare objective values or fitness between current vector and trial vector and select the vector which has better fitness
`public void UpdateBest()`	Update the global best in the population
`public void DisplayBest()`	Show the information of the global best vector on the screen
`public void EvalDispersion()`	Evaluate Dispersion index
`public void EvalStatObj()`	Update statistics related to the objective values of vectors in the population

2.2.3 DE class

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

Attributes:

Name	Type	Description
`Fx/Fn`	Real	The maximal and minimal scale factor value
`coRx/coRn`	Real	The maximal and minimal crossover rate value
`Iter`	Integer	Number of iterations
`nVec`	Integer	Number of vectors
`nDim`	Integer	Dimensions of vectors in a population
`rand`	Random stream	Random object used to generate random number
`Pop`	Population	The population used in the DE algorithm

Methods:

`public virtual void DisplayResult(TextWriter t)`	Write the results of DE to a predefined output file `t`
`public virtual double Objective(DecisionVector p,int trial)`	Evaluate objective value of a vector
`void Evaluate()`	`Objective(DecisionVector p,int trial)` is called to calculate the objective value of all current vectors in `Pop`
`void EvaluateTrial(int M)`	`Objective(DecisionVector p,int trial)` is called to calculate the objective value of each trial vector in `Pop`
`public virtual void InitPop()`	Initialize `Pop` with random particles
`public DE(int nIter, double Fmax, double Fmin, double croRx, double croRn)`	Create a new DE object by passing all DE parameters
`public void Run(TextWriter t, bool debug)`	Perform DE algorithm
`public void SetDimension(int vec, int dim)`	Set Population size and vector's dimensions

The DE algorithm is implemented in *`Run`* method. The basic framework of this algorithm is similar to that of the algorithm introduced in section 1.2. The algorithm initializes a first population with user's predefined parameters such as number of vectors, and dimension of a vector. After an initial population is generated, their fitness (objective value) is evaluated and the global best are determined. Then, the population evolves to a new population by mutation, crossover and selection operation. These processes continue until the stopping condition is met. It is noted that the value of scale factor and crossover rate used this DE

algorithm are set to be varied at each iteration. The dispersion index and statistics collections routines can be called optionally. The C# implementation of this algorithm is presented in Figure 2.5.

```
public void Run(TextWriter t, bool debug)
        {   //DE main iteration

            //double F = Fn;
            double diff = Fx - Fn;
            double coR = coRn;
            double decrCr = (coRx - coRn) / Iter;

            Pop = new Population(nVec, nDim);
            InitPop();
            Evaluate();
            Pop.UpdateBest();
            //Pop.SortingObjective();

            for (int i = 1; i < Iter; i++)
            {
                // F changes every iteration
                double F = Fn + rand.NextDouble() * diff;
                for (int m = 0; m < Pop.Member; m++)
                {
                    Pop.Evole(m, F, coR, rand);
                    EvaluateTrial(m);
                    Pop.Selection(m);
                }

                Pop.UpdateBest();

                if (debug)
                {
                    Pop.EvalDispersion();
                    Pop.EvalStatObj();
                    t.WriteLine("{0} \t {1} \t {2}  \t {3}  \t {4}  \t {5}
\t {6}", i, Pop.posBest, Pop.Vector[Pop.posBest].Objective, Pop.Dispersion,
Pop.AvgObj, Pop.MinObj, Pop.MaxObj);
                }
                coR += decrCr;
            }

        }
```

Figure 2.5: C# implementation of DE algorithm

In DE library, scale factor, *F*, value at each iteration is a random value between *FMax* and *FMin*. As the search progresses, the crossover rate, *Cr*, is set to be linearly increased from *CRn* to *CRx*. In case that the constant value of *F* or *Cr* is required, the minimum and maximum values are set to be equal. When designing this library, our objective is to minimize the users' effort to rewrite the DE algorithm. To solve an optimization problem with DE, 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 DE in ET-library. For more complicated problems, some modifications in DE routines such as mutation strategies, local search, and reinitialization may be modified or added as required. In the next chapter, we introduce some practical applications of DE and also show the flexibility of the design.

CHAPTER 3

APPLICATIONS

This chapter shows how DE Library can be applied to solve Job Shop Scheduling Problem (JSP), a combinatorial optimization problem which has been subject to research for several decades.

3.1 Job Shop Scheduling Problem (JSP)

The job shop scheduling problem (JSP) is a combinatorial optimization problem in which a set of *n* jobs needs to be scheduled on a set of *m* machines in order to optimize a certain criterion, subjected to the constraints that each job has a set of sequential operations through all machines and each operation must be processed on a specified machine with deterministic processing time known in advance. Consequently, there are total of *nm* operations to be sequenced in job shop scheduling problem. An example of JSP with 4 jobs and 3 machines are given in Table 3.1, and a feasible solution of this problem is illustrated in Figure 3.1.

Table 3.1 JSP with 4 jobs and 3 machines

Job	Machine Sequence			Processing Time		
J1	M1	M2	M3	3	2	8
J2	M2	M1	M3	5	3	7
J3	M3	M1	M2	4	4	5
J4	M1	M3	M2	9	7	1

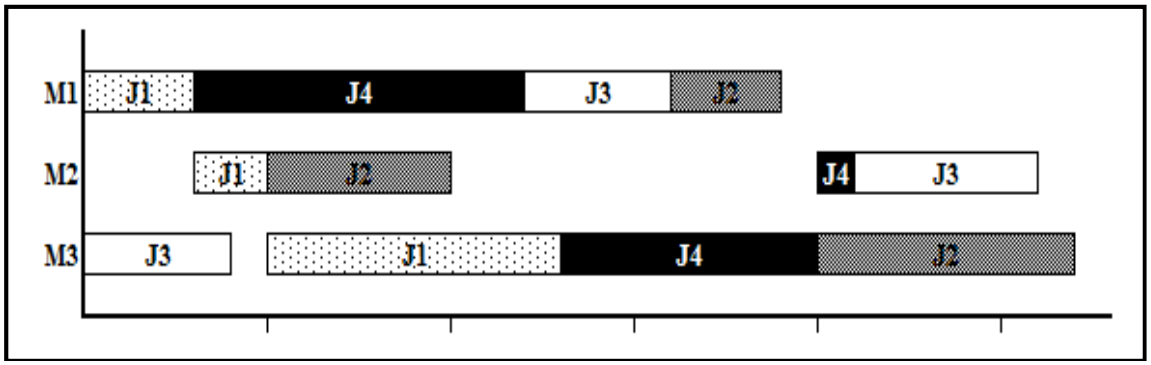

Figure 3.1: Feasible solution for a job shop scheduling problem

Small-size instances of the JSP can be solved to optimal within reasonable computing time by traditional exact algorithms. Unfortunately, when the problem size increases, the computational time of the exact approaches grow exponentially. In some cases, it can take extremely long computational time to find the optimal solution. For these reasons, 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 Application of DE to JSP

This section focuses on the applications of the classic DE with the objective to minimize makespan, C_{max} (maximum completion time of all operations) in JSP. The mathematical model of JSP is provided as follows.

3.2.1 JSP Model

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 Variables:

$s_{j,k}$: The start time of job j on machine k
$y_{j,j',k}$: A binary variable defined as

$$y_{j,j',k} = \begin{cases} 1, \text{if job } j \text{ is before job } j' \text{on machine } k \\ 0, \text{ otherwise} \end{cases}$$

Parameters:

m : The number of machines
n : The number of jobs
$p_{j,k}$: The processing time of job j on machine k
r_j :The ready time of job j
D_j : The due date of job j
M : A large positive number

Objectives:

The objective functions used to evaluate the obtained schedules are formulated differently based on different perspectives. As stated earlier, the objective functions in this JSP example is minimization of makespan, C_{max} , which is defined as follows.

Minimization of makespan

$$C_{max}: minimize\ max\left\{s_{j,k} + p_{j,k}\right\} \tag{3.1}$$

Subjected to the constraints:

Precedence constraints $s_{j,k} + p_{j,k} \leq s_{k'}$ $\forall j, j', k'$ (3.4)

Conflict constraints
$s_{j,k} + p_{j,k} \leq s_{j',k} + M.(1 - y_{j,j',k})$ $\forall j, j', k$ (3.5)
$s_{j',k} + p_{j',k} \leq s_{j,k} + M.y_{j,j',k}$ $\forall j, j', k$ (3.6)

Readiness constraints	$s_{j,k} \geq r_j$	$\forall j, k$	(3.7)
Nonnegative constraints	$s_{j,k} \geq 0$	$\forall j, k$	(3.8)

Where $j, j' = \{1, 2, ..., n\}$ and $k, k' = \{1,2,\ldots, m\}$

3.2.2 Solution Mapping of DE for JSP

It is obvious that the solutions for JSP cannot be directly represented as the *D*-dimensional vector as introduced in chapter 2. For that reason, representation of solution or "solution mapping" is used with encoding and decoding scheme so that the solutions of this problem can be expressed as vectors and each vector can be transformed into a schedule.

3.2.2.1 Encoding

A random key representation scheme (Bean 1994) is employed to encode solutions of the problem to a vector with D dimensions where D is the total number of operations processed on all machines. Consider a JSP example in Table 3.1, each job consists of three operations, and each of these operations must be processed by a specific machine following the operation sequences. The number of dimensions of a solution vector is set to be equal to the total number of all operations which is 12. Suppose that each value in a vector dimension is initially filled with a uniform random number generated in range [0, 1], a random key representation of a solution vector is illustrated in Figure 3.2.

Dimension j^{th}	1	2	3	4	5	6	7	8	9	10	11	12
	0.23	0.15	0.34	0.19	0.71	0.58	0.97	0.46	0.29	0.81	0.65	0.38

Figure 3.2: Random key representation of a solution vector

3.2.2.2 Decoding

Decoding is a process of transforming individuals into a solution. In this research, the permutation of n-repetition of *n* jobs, initially introduced by Bierwirth (1995), is applied with sorting list rule. The decoding procedures are illustrated in Figure 3.3. First, the continuous numbers in vector dimensions of an individual are sorted according to ascending order, and the permutation of repetition of jobs is then applied.

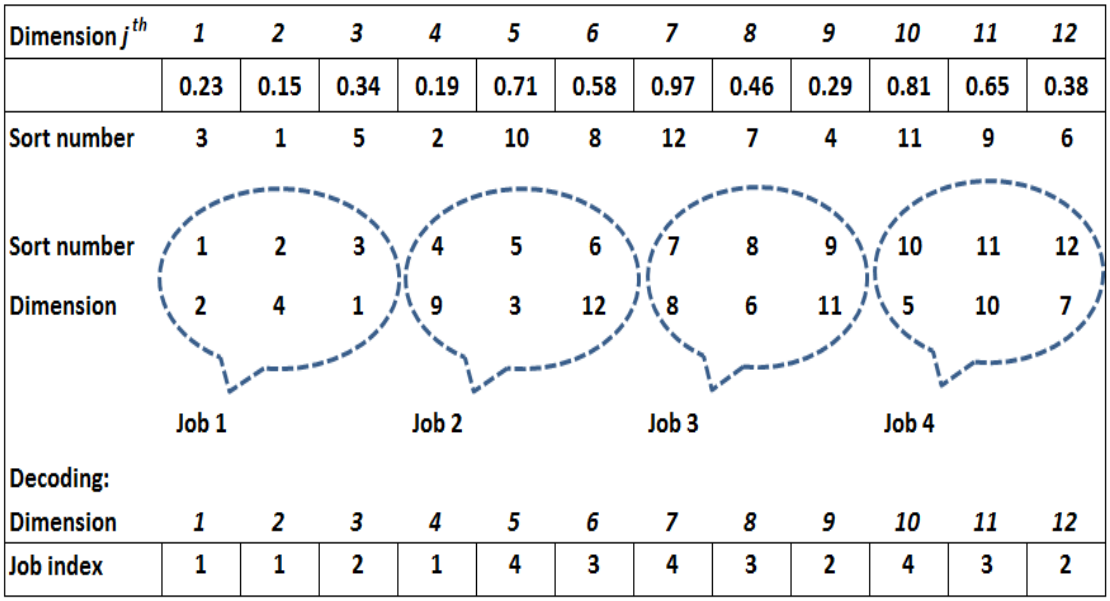

Figure 3.3: Decoding procedures for JSP

The main strength of the permutation of repetition of jobs approach is that any permutation of random key representation always generates a feasible schedule. However, it is possible that some different representations could possibly yield the same schedule. Next, the operation-based approach by Cheng et al. (1996) is employed to generate a schedule as demonstrated in Figure 3.4.

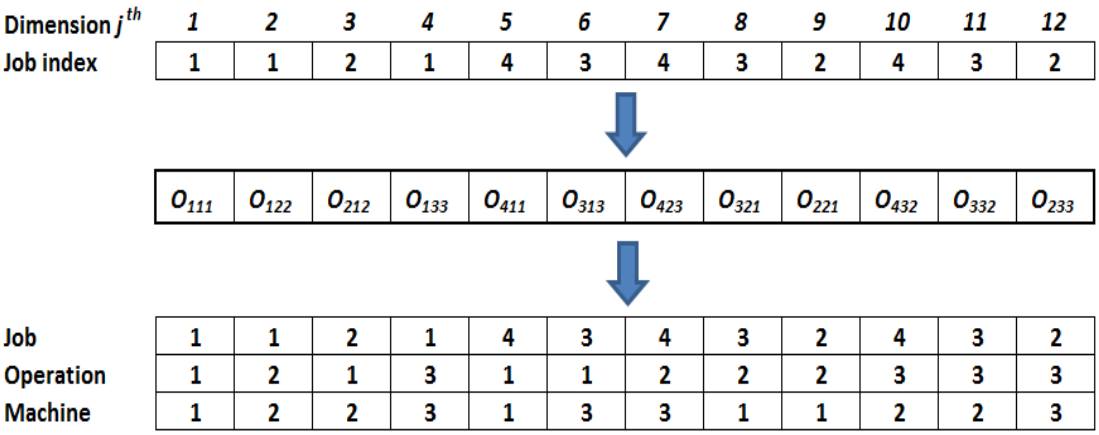

Figure 3.4: The operation-based representation

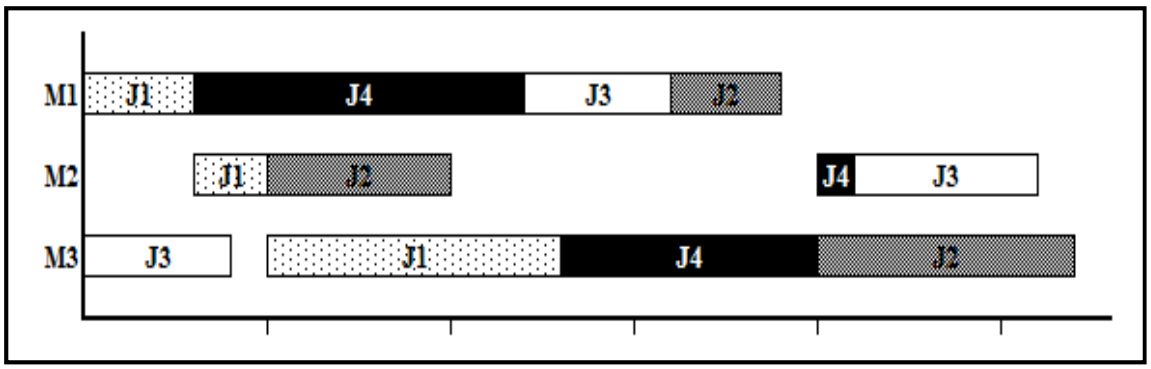

Figure 3.5: An active schedule after the decoding process

The decoded individual is transformed into a schedule by taking the first operation from the sequential orders, then the second operation, and so on until the last operation. In the process of generating a schedule, each operation is assigned to be processed on a required machine in the best available position without delaying other scheduled operations. Thus, this procedure always results in an active schedule as illustrated in Figure 3.5.

The source code for JSP with DE can be found in the manual folder. 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.

The general view of this model is given in Figure 3.6.

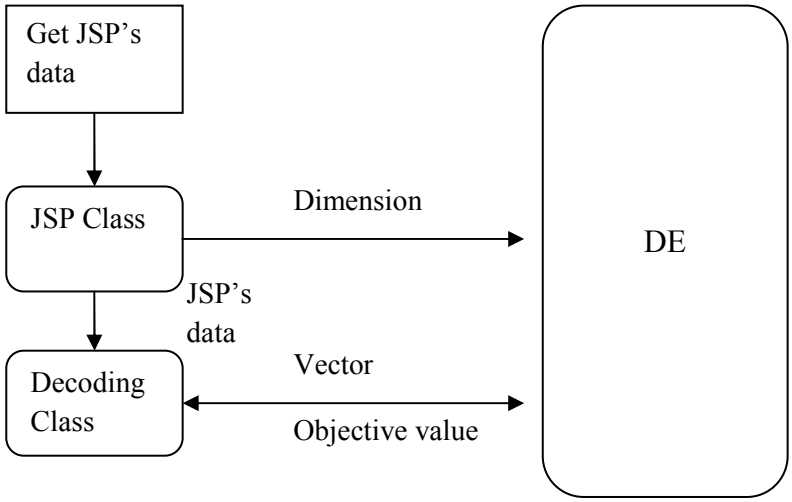

Figure 3.6: DE model for JSP

3.2.3. Reinitialize strategy

During the evolution process, the vectors are often trapped in a deep local minimum which can cause premature convergence of the population. Thus, the reinitialize strategy is applied to diversify the vectors over the search space once again. Consequently, the system could escape from that local trap.

The re-initialize criterion indicates the certain number of iterations to activate the re-initialize process. If the DE algorithm meet the re-initialize criterion, the re-initialize algorithm will start when pre-defined certain iteration number is reached and the procedure will be repeated again every fixed number of iteration. All vectors except the global best vector are selected for re-initialization. As the result, the best solution so far is still reserved in the search process.

3.2.4. Local search strategy

In general, a local search may apply to a certain group of vectors in the population to enhance exploitation of the search, improve the quality of the current solutions, and explore the better solution from a neighborhood of a certain solution. In this study, the neighborhood search adopts the critical block (CB) neighborhood of Yamada and Nakano (1995).

In CB neighborhood concept, a critical path which is the path with the longest length from the first operation on any machine to the last operation on any machine is firstly identified. All operations on the critical path are called the critical operations. Then, a set of consecutive critical operations on the same machine is determined as a critical block. The procedures of the local search in CB neighborhood aim to regulate the search space only on the critical block by moving the critical operations to all possible positions on the block in order to investigate if the solution is improved. Figure 3.7 illustrates the sample of a schedule with a critical path and a set of neighborhood move within a critical block.

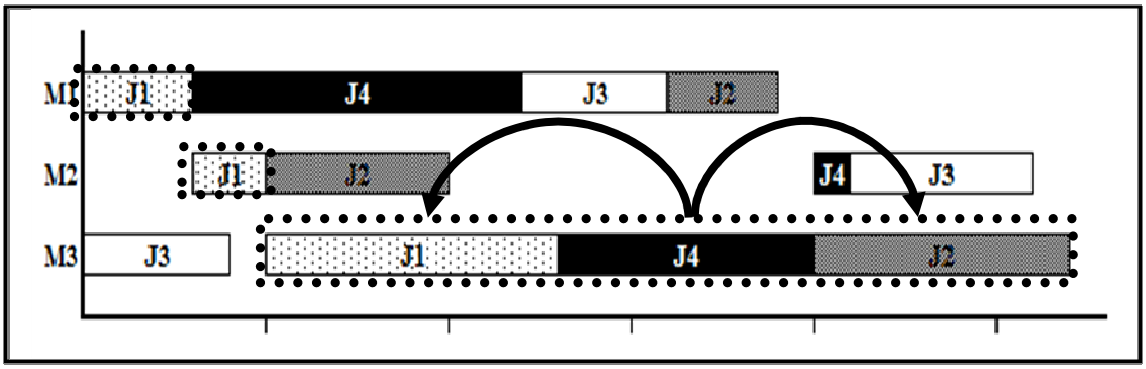

Figure 3.7: CB Neighborhood

As shown in Figure 3.8, a set of neighborhood move is defined within the critical block containing three operations. The operation between the first and the third operations in the block is repositioned to the beginning or the end of that block, and the remaining two operations will be swapped. For each move in the critical block, if the better fitness value is found, the new schedule with its new fitness value is then updated. These procedures are terminated when all moves are completed.

The reinitialize and local search strategy are added to the original algorithm to improve the quality of final solutions by making some attempts to escape from the local optimal. LocalSearchVector(Pop.Vector[j], ref rand) and ReInitPop() are new methods in DE class. At the beginning of search, if the reinitialize or local search condition is met, the population will respectively reinitialize or perform local search on its members instead of performing movement. The C# implementation of DE for JSP is shown as the following.

```
public void Run(TextWriter t, bool debug)
{//DE main iteration
      double diff = Fx - Fn;
      double decrF = (Fx - Fn) / Iter;
      double coR = coRn;
      double decrCr = (coRx - coRn) / Iter;

      Pop = new Population(nVec, nDim);
      InitPop();
      Evaluate();
      Pop.UpdateBest(NB);

      if (debug)
      {
         Pop.EvalDispersion();
         Pop.EvalStatObj();
         t.WriteLine("{0} \t {1}  \t {2}  \t {3} \t  {4}  \t {5} \t  {6}", 0,
Pop.posBest, Pop.Vector[Pop.posBest].Objective, Pop.Dispersion, Pop.AvgObj,
Pop.MinObj, Pop.MaxObj);
       }

       for (int i = 1; i < Iter; i++)
       {
          double F = Fn + rand.NextDouble() * diff;
          bool reinit_locals = false;
          //check condition for LS
          if (((i - startLS) % LSiterval == 0) && (i >= startLS))
            {
               for (int j = 0; j < Pop.Member; j++) //do local search for each member
                   LocalSearchVector(Pop.Vector[j], ref rand);
                   reinit_locals = true;
             }
          //check condition for reinitialize
          if (((i - startReinit) % ReInitIterval == 0) && (i >= startReinit))
             {
                ReInitPop();
                reinit_locals = true;
             }
             if (reinit_locals)//
             {
                 Evaluate();
                 Pop.UpdateBest(NB);
              }
             if (!reinit_locals)
              {
                  for (int m = 0; m < Pop.Member; m++)
                  {
                     Pop.Evole(m, F, coR, rand);
                     EvaluateTrial(m);
                          Pop.Selection(m);
                     }

                       Pop.UpdateBest(NB);
                 }
                  if (debug)
                  {
                      Pop.EvalDispersion();
                      Pop.EvalStatObj();
                      t.WriteLine("{0} \t {1} \t {2}  \t {3}  \t {4}  \t {5}  \t
{6}", i, Pop.posBest, Pop.Vector[Pop.posBest].Objective, Pop.Dispersion,
Pop.AvgObj, Pop.MinObj, Pop.MaxObj);
                  }
                  coR += decrCr;
              }
         }
```

Call reinitialize and local search

Figure 3.8: C# implementation of DE algorithm for JSP

CHAPTER 4

MULTI-OBJECTIVE OPTIMIZATION WITH DE

Previous chapters have demonstrated how DE can be used to solve optimization problems with single objective. However, many real world applications require optimization models to handle more than one objective function. In fact, most real world problems do contain multiple conflicting objectives. As a result, multi-objective optimization (MO) problems have become increasingly attractive to both practitioners and researchers.

4.1 Introduction

Multi-objective optimization (MO) deals simultaneously with more than one objective. The mathematical model for MO problem is given as follow:

Optimize $$f(\vec{x}) = [f_1(\vec{x}), f_2(\vec{x}), \dots, f_k(\vec{x})] \quad (4.1)$$

Subject to:

$$g_i(\vec{x}) \leq 0 \quad i = 1, 2, \dots, n \quad (4.2)$$

$$h_j(\vec{x}) = 0 \quad j = 1, 2, \dots, m \quad (4.3)$$

Where k is the number of objectives in consideration

The solution in MO problems is not determined as only one single solution but rather a set of solutions and the decision about the "best" solution among these solutions mainly depends on the decision maker. Typically, the methods for solving MO problems can be classified into two types: non-Pareto approach and Pareto-based approach. Non-Pareto methods are mostly based on an aggregated weighted approach where multiple objectives functions are given weights apriori and then combined into a single objective function. However, these methods require a set of weights based on preference of the decision maker, which may be subjective due to their limited knowledge about the range of each objective function. More importantly, the fact that only one single solution is found provides the decision maker with inadequate information about the possible trade-offs. Thus, to be more objective, the approaches based on a single aggregated objective function required to be solved several times in order to find sets of solutions corresponding to varying weights, and as the result, these approaches are highly time consuming. On the other hand, weight-free methods such as Pareto-based approaches become more preferable by researchers to solve MO problems because it allows decision makers to simultaneously find the trade-offs or non-dominated solutions on the Pareto front in a single run without prejudice.

4.2 Pareto Optimality Concept

Solution approaches based on Pareto optimality concept aim to search for multiple non-dominated solutions simultaneously. The set of solutions obtained in the Pareto front with an example of two objective functions to be minimized: f_1 and f_2, is illustrated in Figure 4.1.

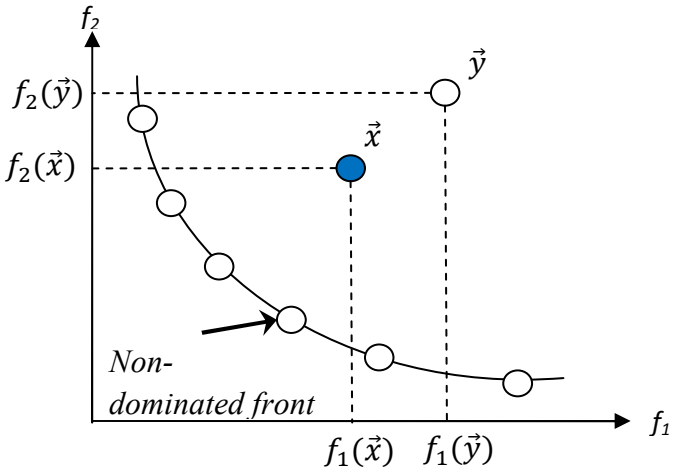

Figure 4.1: $\vec{x} \prec \vec{y}$ and non-dominated solutions in Pareto front

Given two solution $\vec{x}$ and $\vec{y}$, $\vec{x}$ is considered to dominate $\vec{y}$ (denote $\vec{x} \prec \vec{y}$) if and only if $f_i(\vec{x}) \leq f_i(\vec{y})$ for $\forall i$ =1, 2, …, k and $\exists j$ = 1, 2, …, $k \mid f_j(\vec{x}) < f_j(\vec{y})$. As shown in Figure 4.1, for the case that neither $\vec{x} \prec \vec{y}$ nor $\vec{y} \prec \vec{x}$, $\vec{x}$ and $\vec{y}$ are called non-dominated front 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.

4.3 Multi-objective Optimization with DE

The classic DE algorithm was initially developed for optimizing one single objective. In this case, the solution is immediately updated to every single move of a vector when the better solution is found. However, as mentioned earlier that, in MO problems, there is no single solution but rather a set of non-dominated solutions. Thus, the selection based on only one single solution in the classic DE algorithm may not be applicable in multiple objectives environment. Consequently, the classic DE framework needs to be modified when dealing with MO problems. The key components to be considered are listed as follows.

1.) The external archive or storage of non-dominated solutions
2.) The selection of vectors as reference guidance of DE population
3.) Mutation strategy, how to utilize the selected vectors as the search guidance

In multi-objective optimization environment particularly based on Pareto perspective, the searching experience of DE population requires to be stored as a set of non-dominated solutions. The MODE adopts the concept of Elitist structure in NSGA-II (Deb et al., 2002) to store the non-dominated solutions in an external archive, called Elite group. As the search progress, the objective function values of updated vectors must be all sorted by a non-dominated sorting procedure in order to determine the group of vectors in the population which are non-dominated by other vectors. In MODE, instead of applying the sorting procedure to every single move of a vector, the sorting is only performed on the set of newly generated trial vectors after all vectors completed one move. The reason is to reduce computational time. This sorting procedure applies to the group of new solutions and current solutions in the external archive and store only non-dominated solutions into an Elite group. Then, solutions in the Elite group are screened to eliminate inferior solutions. As a result, the

Elite group in the archive contains only the best non-dominated solutions found so far in the searching process of the DE population.

After the Elite group is determined, one of the most challenging issues is the selection of candidates from among the non-dominated solutions in the Elite group as reference guidance in the evolution process of the population. One of the most common techniques is to choose the candidate guidance located in the less crowed regions in order to obtain a better spread of the Pareto front. The first attempt using the concept of crowdedness was introduced in the non-dominated sorting genetic algorithm, NSGA (Srinivas and Deb, 1994) with the use of sharing parameters. However, this approach has been extensively criticized for its computational complexity. An improved version of NSGA, called NSGA-II, was proposed with the introduction of crowding distance (CD) as a spread measurement of non-dominated front. In NSGA-II, a CD value of a particular solution is calculated using the average distance of two points on either sides of this point according to each objective (see Deb et al., 2002 for more details). The advantage of NSGA-II approach is that it is a parameter-free method which does not oblige with a pre-determined setting of sharing parameters as in its first version, NSGA. Coello et al. (2002) presented a PSO algorithm with a geographically-based system to identify the crowded areas. In their approach, the objective space is partitioned into a number of hyper-cubes and each member in the Elite archive is then assigned to one of these hyper-cubes. Once the archive is classified, a hypercube with smallest density is considered and one of its members is randomly chosen as the global guidance.

Finally, the other critical decision in MO problems is how to utilize the selected candidate vectors in mutation strategy as movement guidance toward the Pareto frontier. The existing multi-objective DE algorithms often use only one single solution selected from the Elite group as the search guidance. However, the existence of multiple candidates in the archive offers a large number of options on the movement of vectors. More importantly, the quality of the final solutions will be strongly influenced by the movement behavior adopted by the population. In this research, several mutation strategies in MODE framework are proposed in order to utilize solutions in Elite archive as the movement guidance to obtain the high quality Pareto front. These mutation strategies are discussed in the Section 4.4. The framework MODE algorithm is shown in Figure 4.2.

The procedures of MODE algorithm which takes all aspects discussed above into consideration are described in the Algorithm 4.1. The $Non\_Dominated\_Sort\ (\mathcal{P})$ adopts the sorting procedure as presented in NSGA-II to identify non-dominated solutions. As the search progress, the set of non-dominated solutions will be updated and stored in the Elite group. The fixed number of solutions in the Elite group is implemented in order to reduce the computational time of sorting and updating processes. When the number of non-dominated solutions exceeds the limit, the vectors located in the crowded areas will be selectively removed, thus the Elite group can still maintain the good solutions. The two procedures $Select\_Guidance(\mathcal{E})$ and $Mutation\_operation(Ms)$ are mutation strategy dependent which will be separately discussed in the next section.

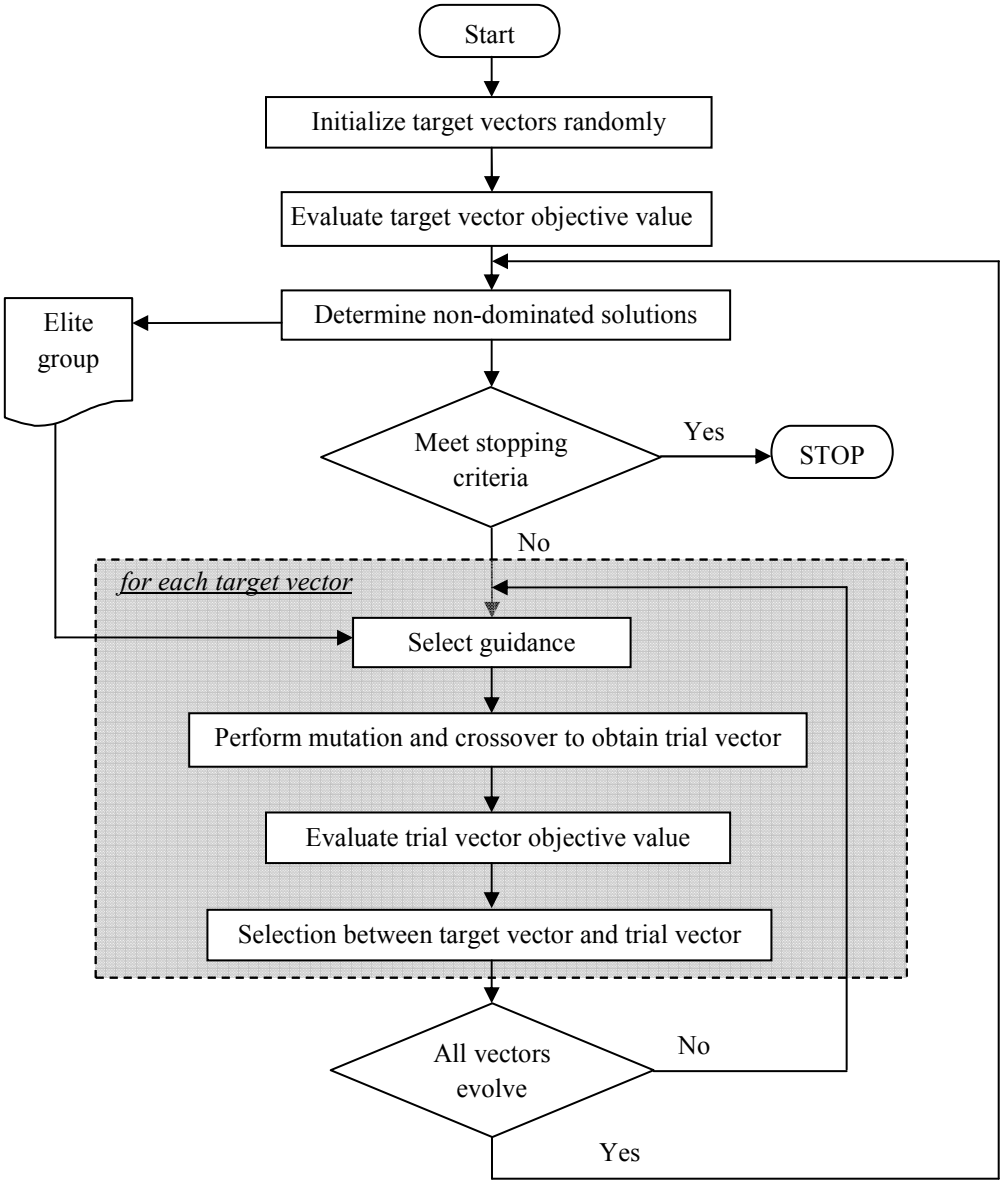

Figure 4.2: Multi-objective differential evolution (MODE) framework

Algorithm 4.1: Procedures of MODE

i. Initialize the DE population, $\mathcal{P}$ with randomly generated *N* target vectors

ii. For each target vector, $\vec{X}_i$, $i = 1, 2, \dots, N$

Evaluate objective function $f_k(\vec{X}_i)$ $\forall k$, $k = 1,2, \dots, K$

iii. $\mathcal{P}* \leftarrow Non\_Dominated\_Sort(\mathcal{P})$

$\mathcal{P}*$ is the set of non-dominated vectors in $\mathcal{P}$

iv. Elite group $\mathcal{E} \leftarrow Non\_Dominated\_Sort(\mathcal{E} \cup \mathcal{P}*)$

v. If stopping criterion is met, end process; otherwise, go to step *iv*.

vi. For each target vector, $\vec{X}_i$, $i = 1, 2, \dots, N$

$Mutation\_operation(Ms) \leftarrow Select\_Guidance(\mathcal{E})$ to obtain a mutant vector $\vec{V}_i$

$Crossover\_operation$ using an equation (1.3) or (1.4) to generate a trial vector $\vec{Z}_i$

Evaluate objective function $f_k(\vec{Z}_i)\ \forall k\ , k = 1,2, \dots, K$

$Selection\_operation$ to determine the survival as the target vector in the next generation

vii. Go to step *iii*

The fact that there is no single best solution on MO environment but rather a set of non-dominated solutions makes the MO problem more complicated. The decision on which non-dominated solutions is better than others cannot be exclusively determined. In the selection process of MODE, there is a situation when the trial vector and target vector are non-dominated. Thus it becomes an issue whether to update the target vector or not. Keeping the current target vector with its positions helps to the algorithm to deeply explore on the local region which may improve the quality of the current solution. On the other hand, it is also desirable to select the trial vector with the new positions to further discover the new area of the front. Since each decision has its own advantages, the MODE algorithm will randomly pick one of them to become the target vector in the next generation.

4.4 MODE Mutation Strategies

As mentioned in the previous section, the external archive called the Elite group is employed as a storage of the non-dominated solutions which can be used to guide vectors in the population. In this chapter, five mutation strategies to utilize the information provided by the Elite group are proposed and discussed as follows.

4.4.1 $Ms1$: Search around Solutions Located in the Less Crowded Areas

The key idea behind this mutation strategy is to generate mutant vectors around solutions on the Pareto front which are located in the less crowded area as shown in Figure 4.3. The goals are to discover more non-dominated solutions and improve the distribution of the current front. In order to measure the crowdedness of a solution vector in the non-dominated front, the crowding distance (CD) proposed in NSGA-II (Deb et al., 2002) is applied and its algorithm is shown in the Algorithm CD. In $Ms1$, after the Elite group is updated, the CD value of each vector in this group is calculated. Vectors with higher CDs are located in less crowded area and they are considered to be better candidates as perturbed vector in this mutation strategy. The pseudo code for $Ms1$ is provided in Algorithm 4.2 where $\mathcal{E}$ is the Elite archive.

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

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

Algorithm 4.2: The pseudo code for $Ms1$

i. Calculate_crowding_distance ($\mathcal{E}$)
ii. Sort $\mathcal{E}$ by decreasing order of crowding distance (CD) values
iii. Randomly select a vector P from top $t\%$ of $\mathcal{E}$
iv. Generate mutant vector (V_i) by Equation (4.4)

$$V_i = P + F * (X_{r1} - X_{r2}) \quad (4.4)$$

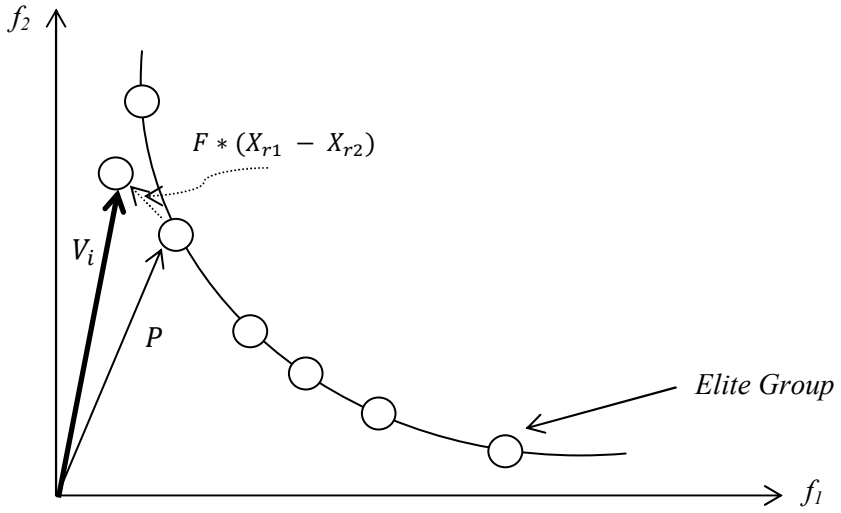

Figure 4.3 Mutant vector generated from $Ms1$ in bi-objective space

4.4.2 $Ms2$: Pull the Current Front to the True Front

This mutation strategy aims to pull the current solutions found so far by the population as close to the solutions on the true Pareto front. For each move, two vectors are randomly selected from Elite group as reference positions to guide the movement of vectors. The implementation of this strategy in bi-objective space is illustrated in Figure 4.4 and its pseudo code is shown in Algorithm 4.3.

Algorithm 4.3: The pseudo code for $Ms2$

i. Randomly select two vectors, R_1 and R_2 from Elite group, $\mathcal{E}$
ii. Generate mutant vector (V_i) by Equation (4.5)

$$V_i = R_1 + F * (R_2 - X_{r1}) \quad (4.5)$$

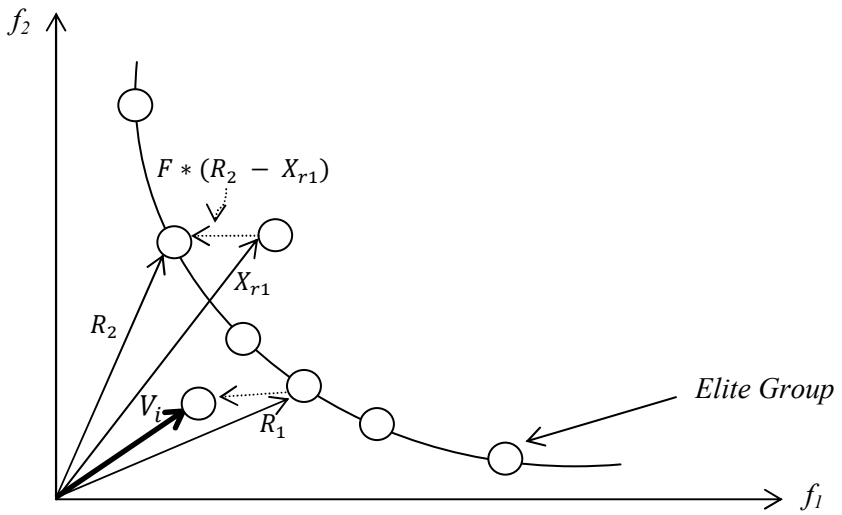

Figure 4.4 Mutant vector generated from $Ms2$ in bi-objective space

4.4.3 $Ms3$: Fill the Gaps of Non-Dominated Front

Inspired by the particle movement strategy in MOPSO, proposed by Nguyen and Kachitvichyanukul (2010), this mutation strategy, $Ms3$, is proposed with an aim to fill the gaps of the non-dominated solutions in order to improve the distribution of the front. The first step of this strategy is to determine the potential gaps in the Elite group. Then, for each objective function $f_k(.)$, the solutions in Elite group are sorted in the increasing order of $f_k(.)$ and the difference of two consecutive vectors is computed. If the difference is larger than predetermined *x%* of the current scale of the non-dominated solutions in the Elite group, it is considered as a gap and the two corresponding vectors are stored as a pair in an unexplored set, $\mathcal{U}$. Then, a pair of vectors is randomly chosen from the set $\mathcal{U}$ and implemented in this mutation strategy to guide vectors in the population to new positions. Figure 4.5 illustrates the direction of the generated mutant vector when the gap is identified and Algorithm 4.4 presents the pseudo code for $Ms3$.

Algorithm 4.4: The pseudo code for $Ms3$

i. Identify the gaps in the Elite group, $\mathcal{E}$
   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,$i+1$) to an unexplored list, $\mathcal{U}$
ii. Randomly select one pair (E_1, E_2) from $\mathcal{U}$
iii. Generate mutant vector (V_i) by Equation (4.6)

$$V_i = (W + r_1) * (E_2 - X_{r1}) + r_2 * (E_1 - E_2) \quad (4.6)$$

Where
r_1, r_2 : a uniform random number in the range [0,1]
W : a constant number which controls the scale difference between E_2 and X_{r1}

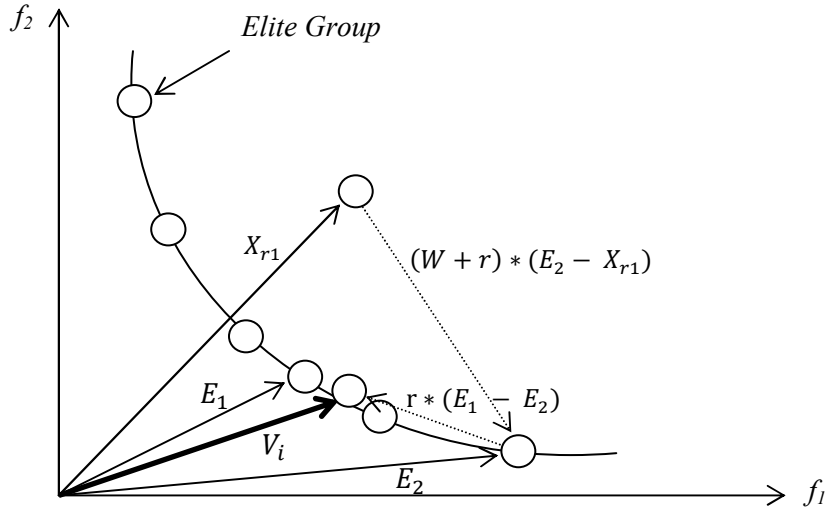

Figure 4.5 Mutant vector generated from $Ms3$ in bi-objective space

4.4.4 $Ms4$: Search toward the Border of the Non-Dominated Front

Different from $Ms1$, $Ms2$, and $Ms3$, the goal of mutation strategy $Ms4$, is to discover the solutions along the current non-dominated front specifically at the boundary in order to increase the spread of non-dominated solutions. In $Ms4$, the vector T_k with the minimum objective function value respective to objective function $f_k(.)$ is set as the reference position for vector movement direction. Two vectors randomly selected from Elite group $\mathcal{E}$ are combined with T_k to form the mutant vectors. The movement direction of the generated mutant vector toward the border of objective function f_1 is illustrated in Figure 4.6. The pseudo code for this algorithm is shown in Algorithm 4.5.

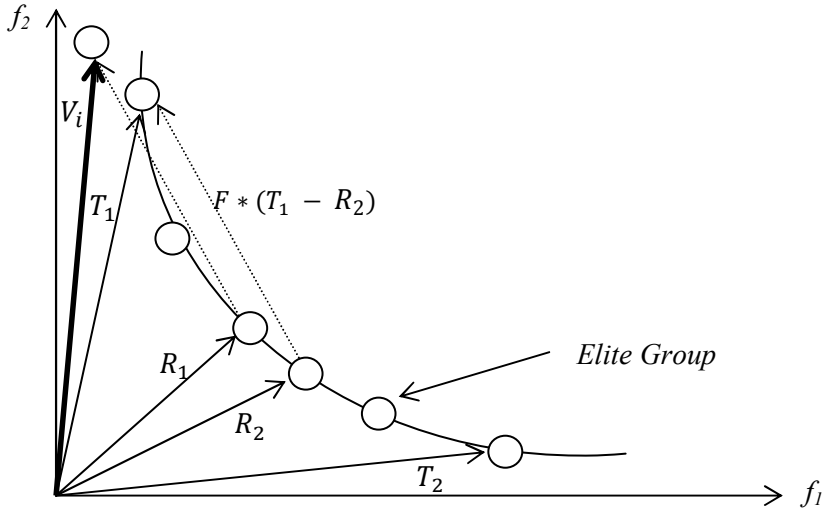

Figure 4.6 Mutant vector generated from $Ms4$ in bi-objective space

Algorithm 4.5: The pseudo code for $Ms4$

i. Select a vector T_k which has the minimum objective function $f_k(.)$ value
ii. Randomly select a vector R_1 from $\mathcal{E}$

iii. Randomly select a vector R_2 from $\mathcal{E}$
iv. Generate mutant vector (V_i) by Equation (4.7)

$$V_i = R_1 + F * (T_k - R_2) \tag{4.7}$$

4.4.5 $Ms5$: Explore Solution Space with Multiple Groups of Vectors

Since different mutation strategies possess their own strengths and perform the search in different ways in order to reach the Pareto optimal front, the purposes of $Ms5$ are to exploit the strengths of various DE mutation strategies and to compensate for the weaknesses of each individual strategy in order to enhance the overall performance and increase the quality of the non-dominated front. In $Ms5$, the DE population is divided into four groups of vectors. Each group executed a particular search strategy with information sharing across other groups, thus the combination of multiple search strategies is embedded in one population.

- Group1 consists of vectors that prefer to search at the less crowded areas ($Ms1$)
- Group2 consists of vectors that aim in the direction of the true Pareto front ($Ms2$)
- Group3 consists of vectors that try to fill the gaps of non-dominated front ($Ms3$)
- Group4 consists of vectors that seeks to find non-dominated solution at the border of the Pareto front ($Ms4$)

In this strategy, the non-dominated solutions found by these four groups of vectors are stored in the common Elite archive which can be accessed by all members. As mentioned above, each group performs different search strategies with own strengths and weaknesses. Since, the mutation strategy $Ms1$ guide vectors to search at the less crowded areas, the quality on the distribution of the non-dominated front can be improved. However, the movement of vectors in this group will mainly depend on how the solutions are distributed in the current Elite archive. Differently, the vectors in group 2 (following $Ms2$) focus on pulling the current non-dominated front toward the true front. Then again, the performance of this strategy deteriorates when only few non-dominated solutions are found since the exploration ability of the vectors is limited. It is important to note that the gaps in the non-dominated front are unavoidable even when the multiple mutation strategies are adopted. Thus, the vectors in group 3 (following $Ms3$) aim to fill these gaps so that more non-dominated solutions are discovered and the final front can have a better distribution. Finally, the task of the vectors in group 4 (following $Ms4$) is to explore the border of the front in order to increase the spread of non-dominated solutions.

4.5 MODE Library

It is noted that the classic DE algorithm needs to be changed to deal with MO problems. Therefore, a new library called M2DE (Multi-strategy Multi-Objective Differential Evolution Algorithm) is developed based on the original framework of DE and includes the suggested modifications proposed in previous sections as shown in Figure 4.2 and Algorithm 4.1. Basically, besides the available routines in the classic DE, 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
Decision Vector	NoObj	Integer	Number of objective functions to be optimized
	Objective	Array of real number	The objective values or the fitness of the vector
	crowdDistance	Real	The crowding distance value which is used to indicate the crowdedness of the current position of the vector
	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 vector (for movement strategy 5)
	MinObj/MaxObj	Array of real number	The minimal and maximal objective value found by the population through the searching process
	AvgObj	Array of real number	The average objective values across all vectors in the population
	movingStrategy	Integer	The index of movement strategy used by the population to explore the Pareto front
	vectorMix	2D array of real number	vectorMix[i, 0] and vectorMix [i, 1] is the accumulative probabilities which are used to indicate which vectors in the population use movement strategy i
	constr	Bool	Indicator of whether the MO problem have constraints or not
	`public void setMovingStrategy (int mS)`	Method	Set the movement strategy of population
	`public void setVectorMix (ArrayList vMix)`	Method	Set the vector mix
	`public void setConstraintMode (bool ctr)`	Method	Set the value of constr
	`private static void AssignUnexploreV(Random rnd, ArrayList USpace, ref DecisionVector E1, ref DecisionVector E2)`	Method	Select a pair of vectors used to indicate the direction to unexplored areas as described in movement strategy 3
	`private static void AssignGlobalV(Random rnd, ArrayList Elist, ref`	Method	Select a vector located in less crowded area (V) and crowded area (S) with the probability topP

	DecisionVector P, ref DecisionVector S, double topP, double topS)		and topS as described in movement strategy 1
	private static void AssignTopBottom(Random rnd, ArrayList Elist, ref DecisionVector T, ref DecisionVector B,ref DecisionVector R1, ref DecisionVector R2, double topP, double topS)	Method	Select a vector located at the border of the current non-dominated front as described in movement strategy 4
	private static void AssignRandomV(Random rnd, ArrayList Elist, ref DecisionVector R1, ref DecisionVector R2)	Method	Select two random vectors from the Elite group as described in movement strategy 2
M2DE	nObj	Integer	The number of objective functions to be optimized
	moveS	Integer	The movement strategy used by the population
	ElististP	Array List	The list of Elite solutions found through the search
	UnexploreSpace	Array List	The list of pairs of vectors which is used to indicate the direction to unexplored areas as described in movement strategy 3
	MaxElististMember	Integer	Upper limit of the ElististP
	vecmix	Array List	The proportion of members in the population assigned to follow each movement strategy
	Constraint	Bool	Indicator of whether the MO problem has constraint or not
	TopEPerc	Real	The percentage of members on the top of the Elite group (in the less crowded areas) which can be randomly selected to become the guidance
	BotEPerc	Real	The percentage of members on the bottom of the Elite group (in the crowded areas) which can be randomly selected to become the guidance
	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 members 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 ElististP, int o)`	Method	Calculate the crowding distance corresponding to each objective function
	`public virtual double[] Objective(DecisionVector p, int trial)`	Method	Evaluate all objective values of a trial vector

4.6 A simple Example of Multi-objective Optimization Problem

In this section, M2DE 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.

$$\text{Minimize} \quad f_1(x) = x^2 \tag{4.8}$$

$$f_2(x) = (x-2)^2 \tag{4.9}$$

Where $x \in [-10^3, 10^3]$

Similar to single objective optimization discussed in chapter 2, we have to determine the dimension of a vector, the method to evaluate the objective values, and the method to initialize the population. In general, M2DE are designed so that problems can be easily formulated without worrying too much about the optimization algorithms. Figure 4.7 shows how a new class is created to solve the problem with M2DE.

```
class spDE : M2DE
{
    public int fx;
    public spDE(int fx, int nVec, int nIter, int nNB, double Fmax, double
Fmin, double croRx, double croRn,  int maxE, int moveStr, ArrayList
vm,double te,double be, double gap)
    :base(nIter, nNB, Fmax, Fmin, croRx, croRn, maxE,moveStr,vm)
       {
        //define problem
          this.fx = fx;
          int dimension=1;
          bool constr = false;
```

```
            int nObj=2;
            SetParameters(nVec, dimension, nObj, constr, te, be, gap);
          //number of vectors, dimension,
          //number of objective (and +1 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 < ((DecisionVector)this.ElististP[0]).NoObj; o++)
          t.Write(((DecisionVector)this.ElististP[i]).Objective[o].ToString
          () + "\t");
           t.WriteLine();
          }
                    t.WriteLine("");
                    t.WriteLine("Result:");
                    t.WriteLine("-------");
         }
         public override double[] Objective(DecisionVector p, int trial)
         {
            double[] obj=new double[p.NoObj];
            Function.SCH_Function(p, obj, trial);
            return obj;
         }
         public override void InitPop()
         {
            for (int i = 0; i < Pop.Member; i++)
            {
               for (int j = 0; j < Pop.Vector[i].Dimension; j++)
               {
                   Pop.Vector[i].CurrentVector[j] = -1000 + 2000 *
rand.NextDouble();
                   Pop.Vector[i].VecMin[j] = -1000;
                   Pop.Vector[i].VecMax[j] = 1000;
                }
                for (int o = 0; o < Pop.Vector[i].NoObj; o++)
                    Pop.Vector[i].Objective[o] = 1.7E308;
             }
          }
}
class Function
{
     public static void SCH_Function(DecisionVector v, double[] obj, int
trial)
         if (trial == 0)
         {
             double var = v.CurrentVector[0];
             obj[0] = Math.Pow(var, 2);
             obj[1] = Math.Pow(var - 2, 2);
         }
         if (trial == 1)
         {
             double var = v.TrialVector[0];
             obj[0] = Math.Pow(var, 2);
             obj[1] = Math.Pow(var - 2, 2);
         }
 }
```

Figure 4.7: 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 M2DE's parameters are defined in the main class as presented in Figure 4.8.

```
class MainClass
      {
        public static void DE(int fx,double[] DEparas, 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(DEparas[0]);
            int noVec = Convert.ToInt32(DEparas[1]);
            double FMin = DEparas[2];
            double FMax = DEparas[3];
            int noNB = Convert.ToInt32(DEparas[4]);
            double COx = DEparas[5];
            double COn = DEparas[6];

            int maxE = Convert.ToInt32(DEparas[9]);
            double TopEp = DEparas[10] / 100;
            double BotEp = DEparas[11] / 100;
            double GapUnexplore = DEparas[12] / 100;

            int moveStrategy = strategy;

            int rSeed = (int)DEparas[17];
            int noRep = (int)DEparas[18];
            // end parameter setting

            if (moveStrategy == 5)
            {
                vMix.Add(0); vMix.Add((double)DEparas[13] / 100);
                vMix.Add(1); vMix.Add((double)DEparas[14] / 100);
                vMix.Add(2); vMix.Add((double)DEparas[15] / 100);
                vMix.Add(3); vMix.Add((double)DEparas[16] / 100);
            }

            // starting time and finish time using DateTime datatype
            DateTime start, finish;
            // elapsed time using TimeSpan datatype
            TimeSpan elapsed;
            ## Write parameters 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 ...
                M2DE GlobalPop = new spDE(fx,noVec, noIter, noNB, FMax,
FMin, COx, COn, maxE, moveStrategy, vMix, TopEp, BotEp, GapUnexplore);
                GlobalPop.SetRSeed(rSeed);
                GlobalPop.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 4.8: C# implementation of M2DE algorithm

The more generalized source code of this example, which includes a convenient interface and a list of test problems, can be found in "\Basic Models\ DE_MutiObjective\ DE_MutiObjective\". This 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 4.9.

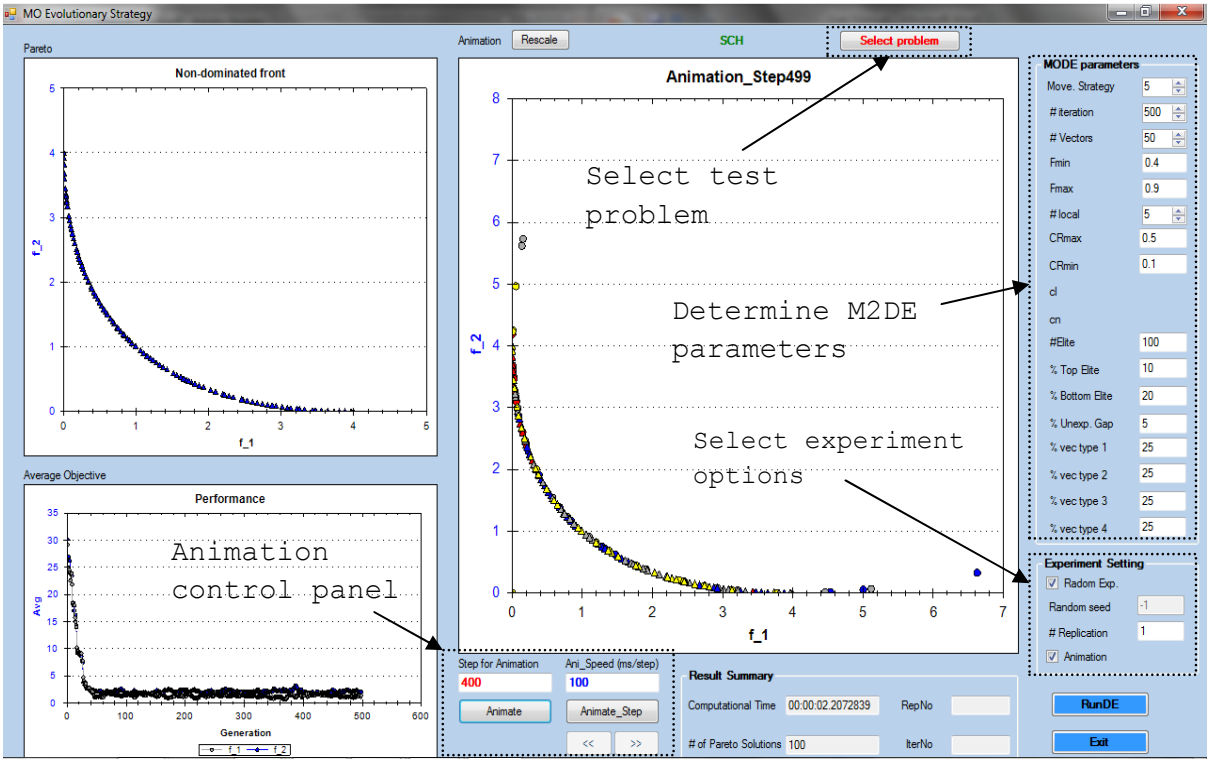

Figure 4.9: Multi-objective optimizer with M2DE

Figure 4.9 shows the interface built for research purpose. The figure on the upper left corner presents the final Pareto front found by the M2DE 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 vector is represented by the circle point. The color of each point is used to identify the type of a vector (in mutation strategy 5). 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
Yellow	Type0	Type0	Type0	Type0	Type0
Gray	N/A	N/A	N/A	N/A	Type1
Blue	N/A	N/A	N/A	N/A	Type2
Red	N/A	N/A	N/A	N/A	Type3

When mutation strategies Ms1-Ms4 are used, all vectors only follow single movement behavior so only one color is used. In mutation strategy Ms5, type 0, 1, 2, 3 indicate the vector in group 1, 2, 3, 4 respectively.

4.7 Multi-objective Optimization in Job Shop Scheduling Problem

In this section, M2DE algorithm is applied to solve Multi-objective Job Shop Scheduling Problem (MOJSP) with the objective to simultaneously minimize makespan and minimize total tardiness of jobs. These two objectives are formulated as the following equation.

Minimization of makespan

$$f_1: minimize\ max\ \{s_{j,k}\ +\ p_{j,k}\} \tag{4.10}$$

Minimization of total tardiness

$$f_2: minimize \sum_{i=1}^{n} \max\ \{0\ ,(\ s_{j,k} + p_{j,k}) - D_j\ \} \tag{4.11}$$

The solution representation used in multi-objective JSP is the same as those used in single-objective JSP in section 3.2.2. Figure 4.10 shows how a new class is created to solve MOJSP with M2DE.

```
    class spM2DE : M2DE
    {
        public spM2DE(int nVec, int nIter, int nNB, double Fmax, double
Fmin, double croRx, double croRn, int dim, JSPdata jd, int maxE, int
moveStr, ArrayList vm, double te, double be, double gap)
        : base(nIter, nNB, Fmax, Fmin, croRx, croRn, dim, jd, maxE,
moveStr, vm)
        {
            JD = new JSPdata(jd.NoJob, jd.NoMc, jd.NoOp, jd.Job,
jd.NoOpPerMc, jd.Machine);
            int nObj = 2;
             base.SetParameters(nVec, dim, nObj,te, be, gap);
        }

        public override void InitPop()
        {
            for (int i = 0; i < Pop.Member; i++)
            {
                for (int j = 0; j < Pop.Vector[i].Dimension; j++)
                {
```

```
                    Pop.Vector[i].CurrentVector[j] = rand.NextDouble();
                }
                //set vector type
                if (Pop.movingStrategy == 5)
                {
                    int ms = 0;
                    for (int k = 0; k < 5; k++)
                        if ((i > Pop.Member * Pop.vectorMix[k, 0]) && (i <=
Pop.Member * Pop.vectorMix[k, 1])) ms = k;
                    Pop.Vector[i].type = ms;
                }

                for (int o = 0; o < Pop.Vector[i].NoObj; o++)
                    Pop.Vector[i].Objective[o] = 1.7E308;
            }

        }

        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 <
((DecisionVector)this.ElististP[0]).NoObj; o++)

t.Write(((DecisionVector)this.ElististP[i]).Objective[o].ToString() +
"\t");
                t.WriteLine();

            }
            t.WriteLine("");
            t.WriteLine("Result:");
            t.WriteLine("-------");
            for (int i = 0; i < this.ElististP.Count; i++)
            {
                t.WriteLine("Elistist {0}", i);
                for (int w = 0; w <
((DecisionVector)this.ElististP[i]).Dimension; w++)
                {
                    t.WriteLine("x({0}) = {1}", w,
((DecisionVector)this.ElististP[i]).CurrentVector[w]);
                }

                FitnessValue myfitness1 = new FitnessValue();

                myfitness1.ScheduleGJSP(JD.NoJob, JD.NoMc, JD.NoOp,
((DecisionVector)this.ElististP[i]).CurrentVector, JD.Job,
((DecisionVector)this.ElististP[i]).Dimension,JD.NoOpPerMc ,JD.Machine,
JD);

                for (int j = 0; j < JD.NoJob; j++)
                {
                    t.WriteLine("");
                    t.WriteLine("J{0}\t Start\t End\n", j + 1);
                    for (int k = 0; k < JD.NoOp[j]; k++)
                    {
                        t.WriteLine("{0} \t {1} \t {2} ", k + 1,
JD.Job[j].Operation[k].StartTime, JD.Job[j].Operation[k].EndTime);

                    }
```

```
                    }

                }

            }
            public override double[] Objective(DecisionVector p, int trial)
            {
                double[] obj = new double[p.NoObj];
                FitnessValue myfitness = new FitnessValue();
                if (trial == 0) myfitness.ScheduleGJSP(JD.NoJob, JD.NoMc,
JD.NoOp, p.CurrentVector, JD.Job, p.Dimension, JD.NoOpPerMc, JD.Machine,
JD);
                if (trial == 1) myfitness.ScheduleGJSP(JD.NoJob, JD.NoMc,
JD.NoOp, p.TrialVector, JD.Job, p.Dimension, JD.NoOpPerMc, JD.Machine, JD);
                objective myObjective1 = new objective();
                obj[0] = myObjective1.Cmax2(JD.NoMc, JD.NoOpPerMc, JD.Machine);
                objective myObjective2 = new objective();
                obj[1] = myObjective2.TotalTardiness(JD.NoJob, JD.NoOp,
JD.Job);
                return obj;
            }
        }
```

Figure 4.10: Implementation of M2DE for MOJSP in C#

The more generalized source code of this example, which includes a convenient interface and a list of test problems, can be found in "\Applications\ DE_MO_JSP\ DE_MutiObjective\". The example of animation feature in MOJSP showing the movement behavior of the solutions is illustrated in the Figure 4.11.

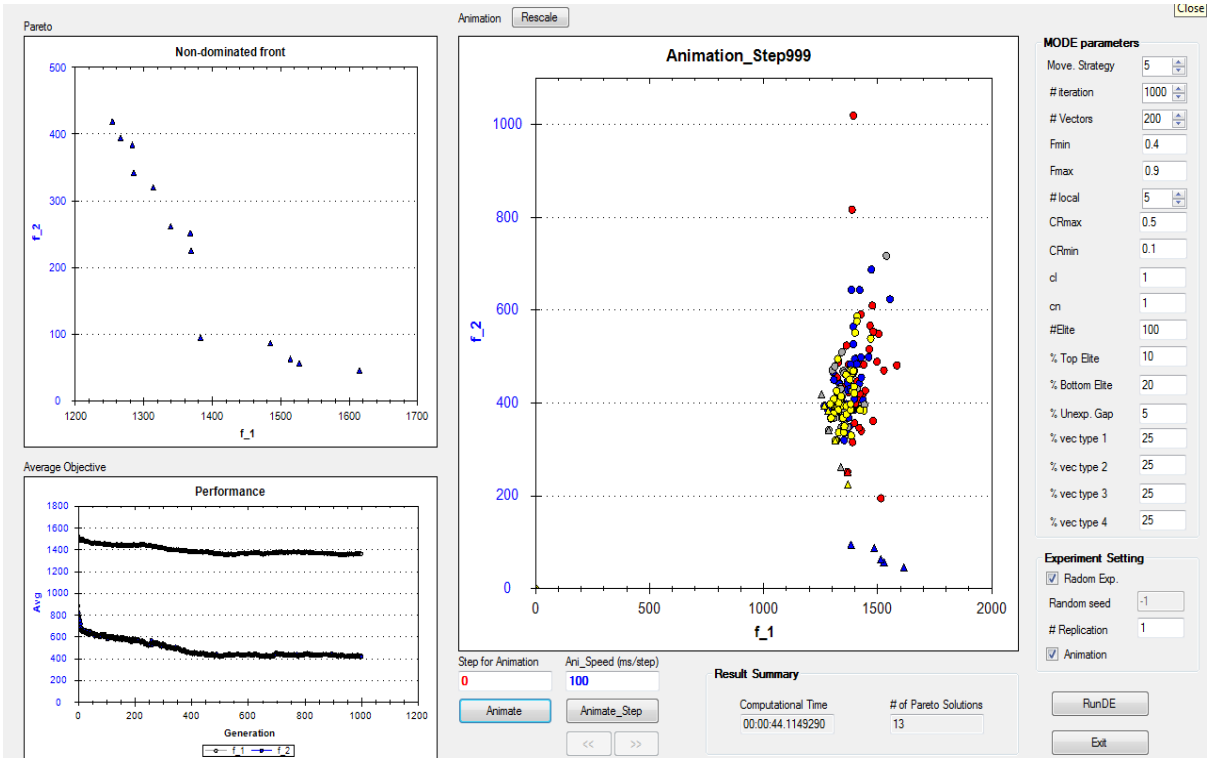

Figure 4.11: Multi-objective optimizer with M2DE for MOJSP