CHAPTER II

FUNDAMENTAL OF EVOLUTIONARY ALGORITHMS

2.1 Introduction

Evolutionary algorithm is a stochastic optimization method based on the principle of natural evolution. The field of investigation that concerns all evolutionary algorithms is known as evolutionary computation. The origin of the evolutionary computation was dated back to 1950s and 1960s, when researchers proposed the use of evolution-inspired algorithms for machine learning and numerical optimization purposes. However, it was not until the 1980s when these methods started to gain increased popularity due to rapid development in computer technology. Since then, the field of evolutionary computation has experienced tremendous growth, and has drawn attention from scientists and researchers of many different disciplines.

Evolutionary algorithms search for the solution of an optimization problem via a simplified model of the evolutionary process observed in nature, where population of organisms evolves over a number of generations striving for survival in a dynamic and highly competitive environment. According to the Darwinian principle of survival of the fittest, those organisms that are better suited to cope with their environmental surroundings are more likely to survive. The field of evolutionary computation finds it inspiration in this fundamental principle of the natural evolution process.

Some of the most popular evolutionary computation techniques are genetic algorithms (GA), evolutionary programming (EP), evolution strategies (ES), and differential evolution (DE). Genetic algorithms (GA) were developed in the early 1960s by Holland [28] at the University of Michigan in Ann Arbor. Holland's primary intention was to study the phenomenon of adaptation as it occurs in nature, and to develop ways in which the mechanisms of natural adaptation could be imported into computer systems. Evolutionary programming (EP) was devised in the early 1960s by Fogel [29] at the University of California in San Diego. This technique was originally intended to simulate intelligent behavior by means of finite-state machines, and was later extended to numerical optimization problems. Evolution strategies (ES) were conceived in the mid-1960s by Rechenberg [30] at the Technical University of Berlin. Rechenberg's original intention was to develop a procedure that would optimize parameters for aerotechnology devices.

Differential evolution (DE) was developed by Storn and Price [17] at the International Computer Science Institute in Berkeley. Despite its simplicity, differential evolution has proven to be a very robust and efficient method for global optimization over continuous spaces.

2.2 Basic Structure of Evolutionary Algorithms

In contrast to traditional optimization methods, evolutionary algorithms operate on a population of individuals, each of which represents a search point in the space of possible solutions of the optimization problems. The algorithmic procedure begins by creating an initial set of contending individuals. Then, these parent solutions are subjected to random variation to create offsprings, i.e. new candidate solutions. This random variation usually includes mutation, i.e. an operation that allows for various attributes of the individuals to be occasionally changed, and recombination, i.e. an operation by which the attributes of two or more existing individuals are combined to create a new individual. Next, competing individuals are evaluated by means of a fitness function whose basic task is to provide a measure of how well adapted each individual of the population is to its environment. Finally, a selection operation determines which individuals will be kept as parents for the following iteration or generation. This iterative process continues, using the selected set of parent solutions, until a particular convergence criterion is satisfied.

For optimization to occur, the selection process must be biased toward those individuals that are better suited to their environment, i.e. by giving them better chances to survive and reproduce. By this means, each successive population will be made of individuals that are collectively fitter than those from previous generations. Figure 2.1 shows the basic structure of a typical evolutionary algorithm [31].

Despite sharing the same overall structure, evolutionary computation techniques differ from a number of aspects. The most noticeable differences between them are related to the type of representation used for candidate solutions (e.g., differential evolution uses a floating-point representation, while classical genetic algorithms operate on binary strings), the type of genetic operations used (e.g., genetic algorithms depend mainly on recombination, whereas evolutionary programming uses mutation as the dominant operation), and the ways each operation is implemented. More information regarding evolutionary computation techniques and their differences can be found in [32-39]. The following section will describe the comprehensive detail

of typical differential evolution (DE) due to its relevance to the work presented in this dissertation.

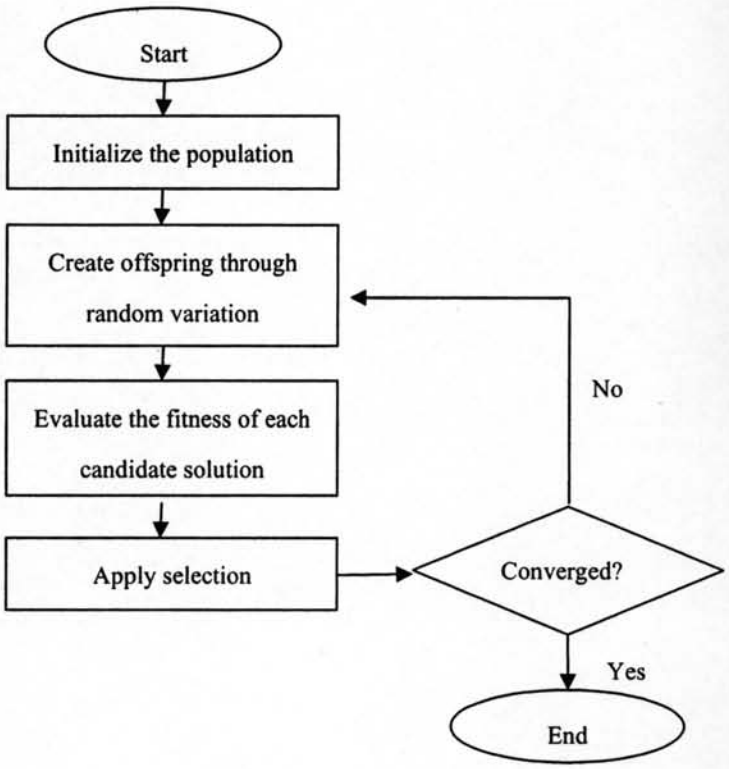

Figure 2.1 Basic structure of a typical evolutionary algorithm.

2.3 Differential Evolution

Differential Evolution (DE) is a new evolutionary algorithm (EA) proposed by Price and Storn [17-20] for real-valued numerical optimization problems. In its canonical form, differential evolution considers unconstrained optimization problems only, except for feasible bounds on the control variables. The general scheme of the DE method resembles that most of other evolutionary algorithms. The advantages of DE are simple structure, easy of use, speed and robustness. Experimental results have shown that DE has good convergence properties and outperforms other well known evolutionary algorithms [17-22]. In this chapter, we will introduce the differential evolution (DE) based on DE/rand/1/bin since this strategy is the most successful and the most widely used strategy. Details of other strategies are described in Appendix A.

Like other evolutionary algorithms, the main design emphasis of DE is real continuous parameters for an unconstraint minimization problem which can be expresses as follows:

$$\min_{X} f(X) \tag{2.1}$$

subject to

$$X_{low} \leq X \leq X_{hi} \tag{2.2}$$

where X represents n-dimensional variables, and X_{low} and X_{hi} are the lower and upper bounds of the variables. The basic algorithm of DE typically consists of four phases, i.e. 1) initialization, 2) mutation, 3) crossover or recombination, and 4) evaluation and selection phases. The DE's algorithm can be described as below.

Phase 1: Initialization

At every generation G, DE maintains a population $P^{(G)}$ of NP vectors of candidate solutions as shown in (2.3)

$$P^{(G)} = [X_1^{(G)},...,X_{NP}^{(G)}] \tag{2.3}$$

The size of the population, NP, is held constant throughout the optimization process. The dimension of each vector of candidate solutions corresponds to the number of decision parameters, n, to be optimized. Therefore,

$$X_j^{(G)} = [x_{1,j}^{(G)},...,x_{n,j}^{(G)}]^T, \quad j = 1,...,NP \tag{2.4}$$

In order to establish a starting point for the optimization process, the population must be initialized. Typically, each decision parameter in every vector of the initial population is assigned a randomly chosen value from within its corresponding feasible bounds:

$$x_{i,j}^{(0)} = x_i^{(0)} + U_i(x_i^{max} - x_i^{min}); \; i = 1,...,n, j = 1,...,NP \tag{2.5}$$

where U_i denotes a uniformly distributed random number within the range [0,1] for each value of i; x_i^{max} and x_i^{min} are the upper and lower bounds of the ith decision parameter respectively. Once every vector of the population has been initialized, its corresponding fitness value is calculated and stored for future reference.

Phase 2: Mutation

The population of the following generation, $P^{(G+1)}$, is created by successively applying mutation, recombination, also known as crossover, and selection operations to the parameter vectors of the current population $P^{(G)}$. Once every generation, each parameter vector of the current population becomes a target vector. For each target vector, the mutation operation produces a new parameter vector, called a mutant vector, by adding the weighted difference between two randomly chosen parameter vectors to a third parameter vector, also random chosen. The crossover operation generates a trial vector, by mixing the parameters of the mutant vector with those of the target vector. If the trial vector obtains a better fitness value than the target vector, then the trial vector replaces the target vector in the following generation. This iterative procedure continues until each parameter vector in the current population has served once as the target vector. The overall optimization process is terminated until a specified convergence criterion is met.

At every generation G, each vector in the population has to serve once as a target vector. For each target vector $X_i^{(G)}$, a mutant vector $X_j^{'(G)}=[x_{1,j}^{'(G)}, \ldots, x_{n,j}^{'(G)}]^T$ is generated according to the following expression:

$$X_j^{'(G)} = X_a^{(G)} + F(X_b^{(G)} - X_c^{(G)}),\ j = 1,\ldots,NP \tag{2.6}$$

where a, b, and c are randomly chosen indices, such that $a, b, c, j \in \{1,\ldots,NP\}$ and $a \neq b \neq c \neq j$. It should be noted that new values for a, b, and c have to be generated for each value of j, i.e. for each individual in the population. F is a user defined constant (known as the mutation factor), which is typically chosen from within the range $0 < F \leq 1.2$ [17-19]. The task of this is to control the amplification of the vector difference $(X_b^{(G)} - X_c^{(G)})$. In the event that mutation causes a parameter to exceed its feasible bounds, the value of the parameter is set to the corresponding violated bound.

Phase3: Crossover or Recombination

In order to increase the diversity among the mutant parameter vectors, crossover is introduced. To this end, a trial vector $X_j^{''(G)}=[x_{1,j}^{''(G)}, \ldots, x_{n,j}^{''(G)}]^T$ is created from the components of each mutant vector $X_j^{'(G)}$ and its corresponding target vector $X_j^{(G)}$, based on a series of n-1 binomial experiments [20, 21] of the following form:

$$X_{i,j}^{''(G)} = \begin{cases} X_{i,j}^{'(G)}, \text{ if } U_i \le CR \text{ or } i = q \\ X_{i,j}^{(G)}, \text{ otherwise} \end{cases} ; i = 1,...,n,\ j = 1,...,NP \tag{2.7}$$

where U_i denotes a uniformly distributed random number within the range of $0 \le U_i < 1$ for each value of i. CR is a user-supplied parameter (known as the crossover constant), which is usually chosen from within the range of $0 \le CR \le 1$. This constant controls the probability that a trial vector parameter will come from the mutant vector, instead of from the target vector. Note that when CR is equal to 1, the trial vector $X_j^{''(G)}$ is the same as the mutant vector $X_j^{'(G)}$. Thus, the possibility of $X_j^{''(G)}$ and $X_j^{(G)}$ from being exactly equal to each other is avoided. It should be noted that a new random value for q has to be generated for each value of i. Once the composition of each trial vector is determined, its corresponding fitness value is calculated and stored for future reference.

Phase4: Evaluation and Selection

In order to decide whether the trial vector should become a member of the next generation, its fitness value is compared to that of its counterpart in the current population (i.e., its correspondent target vector). The fitter of the two vectors is then allowed to advance into the next generation. That is:

$$X_j^{(G+1)} = \begin{cases} X_j^{''(G)}, \text{ if } f(X_j^{''(G)}) \le f(X_j^{(G)}) \\ X_j^{(G)}, \text{ otherwise} \end{cases} ; j = 1,...,NP \tag{2.8}$$

Note that by using this selection criterion, all the individuals of the next generation are as good as or better than their counterparts in the current generation. This optimization process continues successively whenever convergence criteria e.g. maximum numbers of generation of the population is met.

Generally, DE is easy to work with, since it uses only three control parameters, i.e. the population size NP, the mutation factor F, and the crossover constant CR. In most cases, those control parameters are obtained by a trial-and-error process before starting implementation of optimization problems. Price & Storn [20-22] have given some simple rules for choosing key parameters of DE for any given application. Normally, NP should be about 5 to 10 times the dimension, i.e. number of control variables in optimization problem. As for F, it lies in the range 0.4 to 1.0. Initially F = 0.5 can be tried then F and/or NP is increased if the population converges

prematurely. A good first choice for CR is 0.1, but in general CR should be as large as possible [17-19]. The general flowchart of the DE algorithm is shown in Figure 2.2.

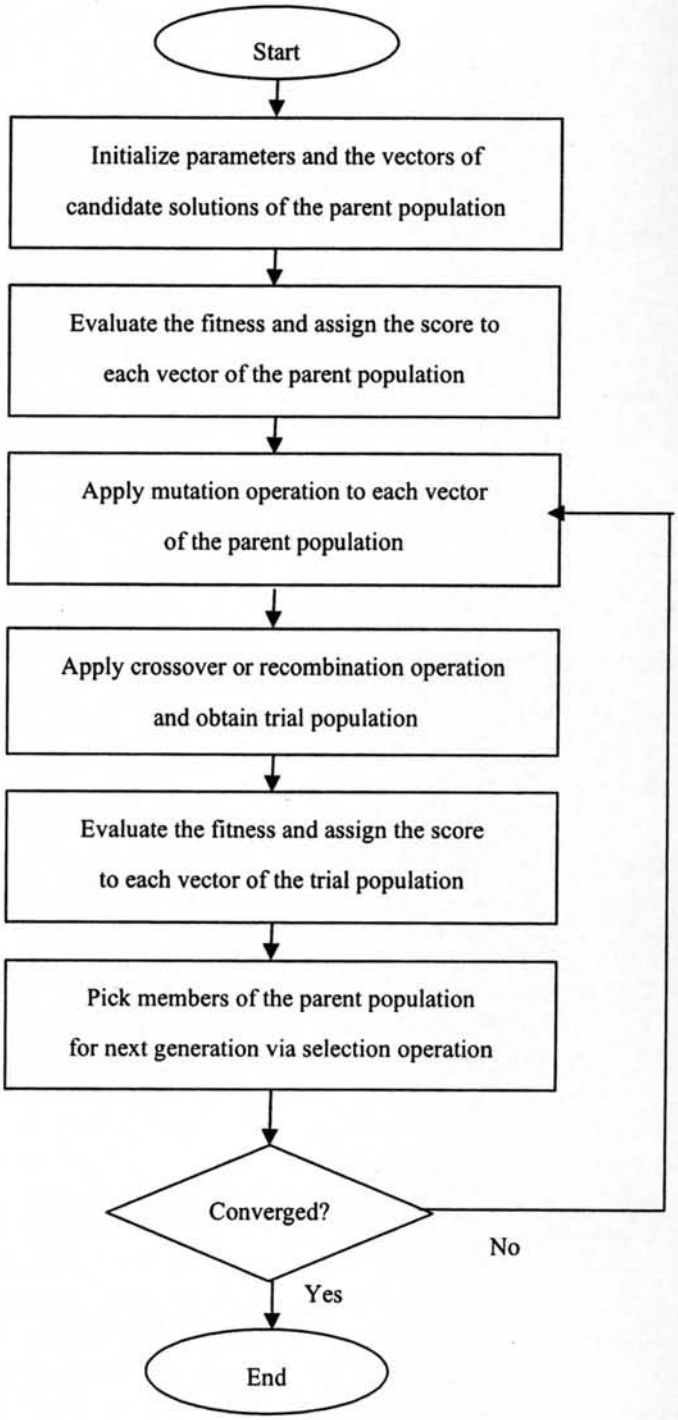

Figure 2.2 General flowchart for the DE algorithm.

From the previous discussion, we can summarize the difference between DE and GA as follows [40-41]:

1) DE uses real number representation while conventional GA uses binary, although GA sometimes uses integer or real number representation as well [41].

2) In GA, two parents are selected for crossover and the child is a recombination of the parents. While in DE, three parents are selected for crossover and the child is perturbation of one of them.

3) The new child in DE replaces a randomly selected vector from the population only if it is better than their parent. In conventional GA, children replace their parents with some probability regardless of their fitness.

In the next section, we will explain the procedure of the DE through a simple optimization problem. The example will depict how the genetic operations of the DE method, i.e. mutation, crossover, and selection operation, can be deployed to generate the member of the population for each successive generation.

2.4 Differential Evolution - Example

The optimization problem to be considered in this example is shown below.

$$\begin{aligned} &\textit{Min} && f(x) = x_1 + x_2 + x_3 + x_4 \\ &\textit{Subject to} && 0 \le x_k \le 1,\ k = 1,\ldots,4 \end{aligned} \tag{2.9}$$

where the number of the decision parameter is $D = 4$. The main objective of this example is to illustrate the procedure of the DE; i.e. the mutation, crossover and selection operation. To keep the example manageable, a population size (NP) of 6 has been chosen. Each decision parameter in every vector of the parent population is initialized within its corresponding feasible bounds, according to (2.5). Once every vector of the parent population has been properly initialized, its corresponding fitness value is calculated through its fitness function f(x) as follows:

$$X_1 = [0.68, 0.89, 0.04, 0.06]^T, \quad f(X_1) = 1.67 \tag{2.10}$$

$$X_2 = [0.92, 0.92, 0.33, 0.58]^T, \quad f(X_2) = 2.75 \tag{2.11}$$

$$X_3 = [0.22, 0.14, 0.40, 0.34]^T, \quad f(X_3) = 1.10 \tag{2.12}$$

$$X_4 = [0.12, 0.09, 0.05, 0.66]^T, \quad f(X_4) = 0.92 \tag{2.13}$$

$$X_5 = [0.40, 0.81, 0.83, 0.12]^T, \quad f(X_5) = 2.16 \tag{2.14}$$

$$X_6 = [0.94, 0.63, 0.13, 0.34]^T, \qquad f(X_6) = 2.04 \tag{2.15}$$

Mutation:

The mutation operation of DE involves two main steps. First, one of the members of the parent population has to be chosen as a target vector. Then, three randomly selected vectors from the parent population (different from each other and the target vector) are used to generate a mutant vector according to (2.6). For this example, let us assume that the chosen target vector is X_1 and the three randomly selected vectors are X_2, X_4, and X_6 respectively. Then, the mutant vector associated with target vector X_1 is generated from:

$$X_1' = X_6 + F(X_2 - X_4) \tag{2.16}$$

Assuming the scaling mutation vector F = 0.80, thus the corresponding mutant vector is:

$$X_1' = [1.58, 1.29, 0.35, 0.28]^T \tag{2.17}$$

From the result, we find that the first and second parameters of the mutant vector exceeded their upper bounds. Therefore, the parameters are changed to their corresponding upper bounds as follows:

$$X_1' = [1.00, 1.00, 0.35, 0.28]^T \tag{2.18}$$

Crossover:

The crossover operation of DE involves the creation of a trial vector, which contains decision parameters from both the target vector and the mutant vector. The trial vector is created according to (2.7). Each parameter in the trial vector is determined by comparing a uniformly distributed random number *U* to *CR* (Crossover Constant). If the random number is lower than *CR*, the trial parameter comes from the trial vector; otherwise, the parameter comes from the target vector. It should be noted that the trial vector always gets, at least, one parameter from the mutant vector. In order to determine which decision parameter from the mutant vector is to be transferred unconditionally to the trial vector, a random number q is chosen from within the set of decision parameter indices. For this example, let us assume that *CR* and *q* are 0.50 and 1 respectively. Additionally, we assume that the uniformly distributed random numbers for the 2^{nd}-4^{th} parameters are 0.45, 0.10 and 0.20 in turn. Since all of the generated values are lower than *CR*; therefore, the trial vector with its fitness value is:

$$X_1'' = [1.00, 0.89, 0.04, 0.06]^T, \qquad f(X_1'') = 1.99 \tag{2.19}$$

Selection:

The new member for the next generation is created by comparing the fitness value between the target vector and the trial vector according to (2.25). The vector which obtains fitter value (in this case lower value for minimization) is selected as a new parent in the next generation. Therefore, the target vector $X_1 = [0.68, 0.89, 0.04, 0.06]^T$ is selected as a new parent in the next generation.

The new population for the next generation is completed by successively applying the mutation, crossover, and selection operations to the current population as already shown in the preceding explanation. The overall optimization process is terminated according to predetermined termination criterion is reached e.g. maximum numbers of generation of the population. Figure 2.3 illustrates the overall process of this example which finally the solution converges to [0, 0, 0, 0].

In the following section, methodology for handling constraints, i.e. the penalty function methods, and the augmented lagrange multiplier method (ALM) will be presented. Since in real world applications, optimization problems are coupled with complex nonlinear constraints. Then, the proposed algorithm called a self-adaptive differential evolution with augmented lagrange multiplier method (SADE_ALM) will be described later.

2.5 Constraints Handling

Generally, a nonlinear constrained minization problem can be expressed as follows:

$$\underset{X}{Min}\ f(X) \tag{2.20}$$

subject to

$$h_k(X) = 0,\ k = 1,\ldots,l \tag{2.21}$$

$$g_j(X) \le 0,\ j = 1,\ldots,m \tag{2.22}$$

$$X_{low} \le X \le X_{hi} \tag{2.23}$$

where X represents n-dimensional variables, $h_k(X)$ and $g_j(X)$ stand for l-equality and m-inequality constraints, respectively, and X_{low} and X_{hi} are the lower and upper bounds of the variables.

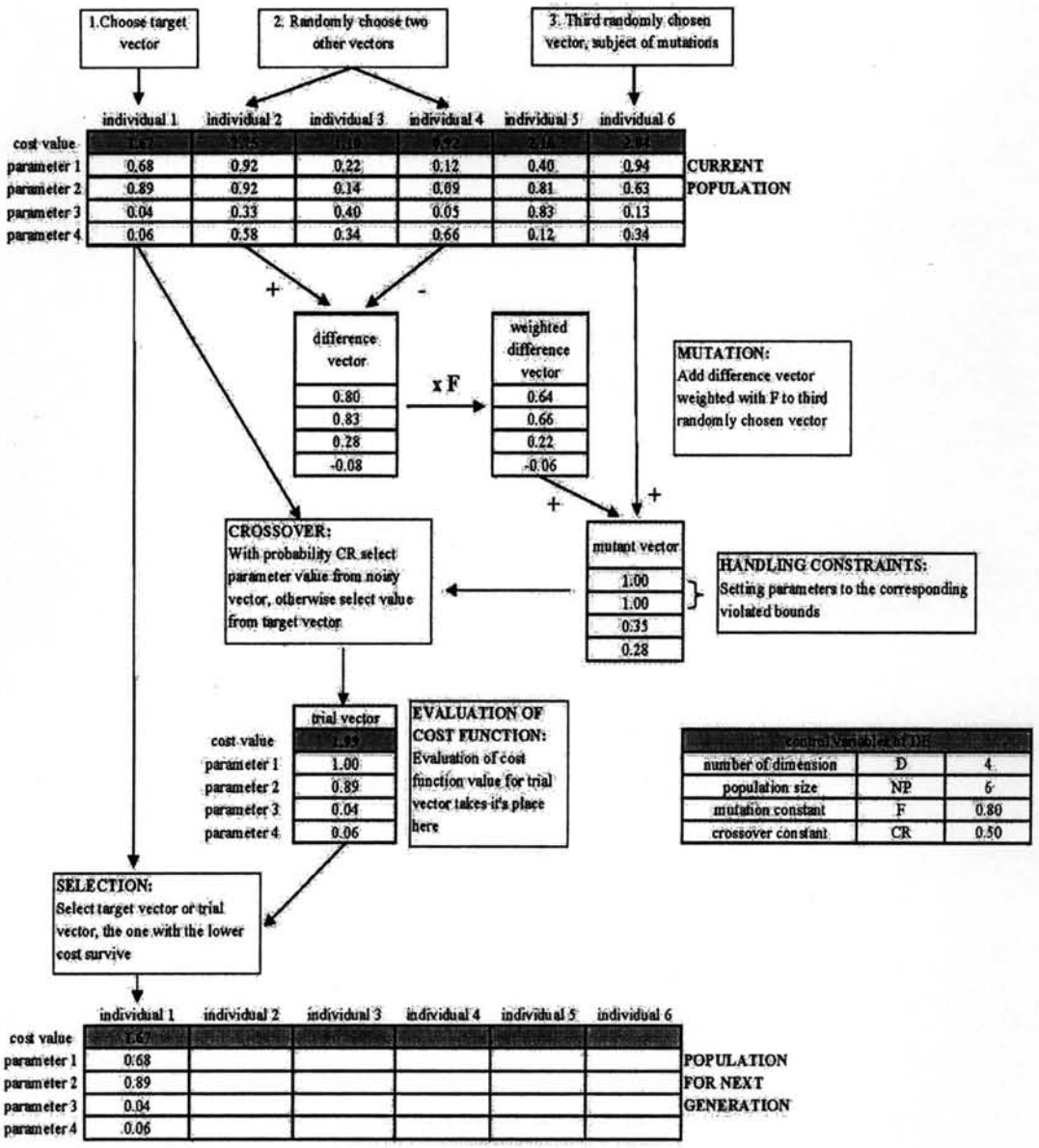

Figure 2.3 Illustration of a simple example of DE where the objective function is min $f(x) = x_1 + x_2 + x_3 + x_4$, subject to $0 \le x_k \le 1$, k=1,…,4.

Michalewicz *et al.* [42] surveyed and compared several constraint-handling techniques used in evolutionary algorithms, and revealed that the penalty function method is the most popular methods for handling constraints, due to its simple concept and convenience to implementation. Such techniques convert the primal constrained problem into an unconstrained problem by penalizing those solutions which are infeasible as follows:

$$L_a = f(X) + \sum_{k=1}^{l} \alpha_k h_k^2(X) + \sum_{j=1}^{m} \beta_k \{max\left[g_j(X), 0\right]\}^2 \qquad (2.24)$$

where α_k and β_k are the positive penalty parameters for equality and inequality constraints respectively, and L_a is the pseudo-objective function or auxiliary function.

From (2.24), it can be seen that the penalty term associated with equality and inequality constraints is added to the objective function. As a result, the penalty term reflects violation of the constraints and assigns the high cost of the penalty function to a candidate individual that is far from the feasible region. When we apply differential evolution (DE) or other evolutionary algorithms (EAs) to solve the penalty problem, any candidate individuals that violate the constraints will impose a high cost or inherit poorer fitness, and find it difficult to survive.

Although the penalty function methods are easy to implement, the main limitation of such methods is the degree to which each constraint is penalized. The penalty function methods do suffer from the complication that as the penalty parameters are increased toward infinity; the structure of the unconstrained problem becomes increasing ill-conditioned. Therefore, each unconstrained problem becomes more difficult to solve, which has the effect of slowing the convergence rate of the overall optimization process. On the other hand, if the penalty parameters are too small, the constraint violation will not impose a high cost on the penalty function. Thus the optimal solution based on the penalty function method may not be feasible. Therefore, choosing appropriate penalty parameters is not a convenient task [43-45].

In contrast to the penalty function methods, the augmented lagrange multiplier method (ALM) can be employed to handle equality/inequality constraints without those difficulties. Essentially, this approach is a penalty-like method, in which the auxiliary function or the augmented lagrange function is obtained by combining the ordinary Lagrangian function with a quadratic penalty functions. Therefore, each unconstrained problem is a minimization of an augmented lagrange function, which has the following form [44-45]:

$$L_a = f(X) + r_h \sum_{k=1}^{l} h_k(X)^2 + r_g \sum_{j=1}^{m} \left\{ max\left[g_j(X), -\frac{\beta_j}{2r_g} \right] \right\}^2 + \sum_{k=1}^{l} \lambda_k h_k(X) + \sum_{j=1}^{m} \beta_j \left\{ max\left[g_j(X), -\frac{\beta_j}{2r_g} \right] \right\} \tag{2.25}$$

where r_h and r_g are the positive penalty multipliers, and the corresponding lagrange multipliers λ_k, β_k are associated with equality and inequality constraints, respectively.

After solving unconstrained problem, the lagrange multipliers and the penalty parameter are updated in order to improve the convergence of the algorithm. The lagrange multipliers are typically updated as follows:

$$\lambda_k^{i+1} = \lambda_k^i + 2r_h h_k\left(X^*\right) \tag{2.26}$$

$$\beta_j^{i+1} = \beta_j^i + 2r_g\left\{max\left[g_j\left(X^*\right), -\frac{\beta_j^i}{2r_g}\right]\right\} \tag{2.27}$$

From (2.26) and (2.27), the multipliers are deterministically updated using the constraint functions evaluated from the previous solution of the unconstrained minimization problem, X^*. Then, each of the penalty parameter is increased by a constant rate until it reaches the predetermined maximum value as shown in (2.28) and (2.29).

$$r_h^{i+1} = \begin{cases} c_h \times r_h^i, & if\ r_h^i \le r_{h,max} \\ r_{h,max}, & otherwise \end{cases} \tag{2.28}$$

$$r_g^{i+1} = \begin{cases} c_g \times r_g^i, & if\ r_g^i \le r_{g,max} \\ r_{g,max}, & otherwise \end{cases} \tag{2.29}$$

where c_h and c_g are positive constant increasing rate, and $r_{h,max}$ and $r_{g,max}$ are the maximum penalty multiplier corresponded with equality and inequality constraints, respectively.

Under appropriate conditions, it can be shown that the ALM approach will converge without having to increase the penalty parameters to a very large value. Therefore, this algorithm is likely to overcome the ill-conditioning associated with classical penalty methods, and consequently, it offers the possibility of faster convergence rates [44-45].

2.6 The Self-adaptive Differential Evolution with Augmented Lagrange Multiplier Method

The self-adaptive differential evolution with augmented lagrange multiplier method (SADE_ALM) is an enhanced version of the traditional differential evolution by integrating two strategic parameters of DE, i.e. the mutation factor (F) and the crossover constant (CR) as additional control variables. As a result, the F and CR can be dynamically self-adaptive

throughout the evolutionary process to increase the capability of avoiding local optimality trapped. Since tuning DE's parameters, i.e. *F*, *CR*, and *NP* is not a easy task due to complex relationship among parameters, even if extensive preliminary experiments were conduct before hand, the optimal parameter settings may never be found and possibly converge to a local optimum [42]. With the proposed method, the trial-and-error process to determine the optimal parameters setting is also reduced. The augmented lagrange multiplier method (ALM) is applied to handle inequality constraints instead of the traditional penalty function method in order to avoid the ill-condition of the augmented lagrange function and the dependency of the penalty parameters which impede the solution search capability. Additionally, the most feasible elitism is also employed to increase the speed of convergence [1, 2].

The algorithm of SADE_ALM consists of two iterative loops, i.e. the inner loop and the outer loop. The inner loop solves the unconstrained minimization problem through the augmented lagrange function L_a using self-adaptive differential evolution (SADE). Figure 2.4 shows the chromosome structure of SADE. It can be seen that the mutation factor (*F*) and the crossover constant (*CR*) are embedded as additional control variables in the first and second positions of the *n*-dimensional parent vector X_j respectively. After the unconstrained minimization problem has been solved, the outer loop will update the lagrange multipliers βs and the penalty parameter r_g by the ALM method to create the new augmented lagrange function L_a. The algorithm is then repeated until a termination criterion, i.e. maximum number of iterations or convergence of the optimal solution, is reached. The flowchart of the SADE_ALM is shown in Figure 2.5.

	1	2	3	4		n+2
X_j	F_j	CRj	x_{1j}	x_{2j}	...	x_{nj}

Figure 2.4 Chromosome structure of SADE.

2.6.1 Inner loop iteration

The inner loop of SADE_ALM comprises 6 steps as described hereafter.

Step 1 (Initialization of SADE). Set maximum iteration of inner loop (N_i), convergence tolerance ($\varepsilon_{\Delta X}$), and then create the initial population sized *NP* associated with their lower and upper limits as follows:

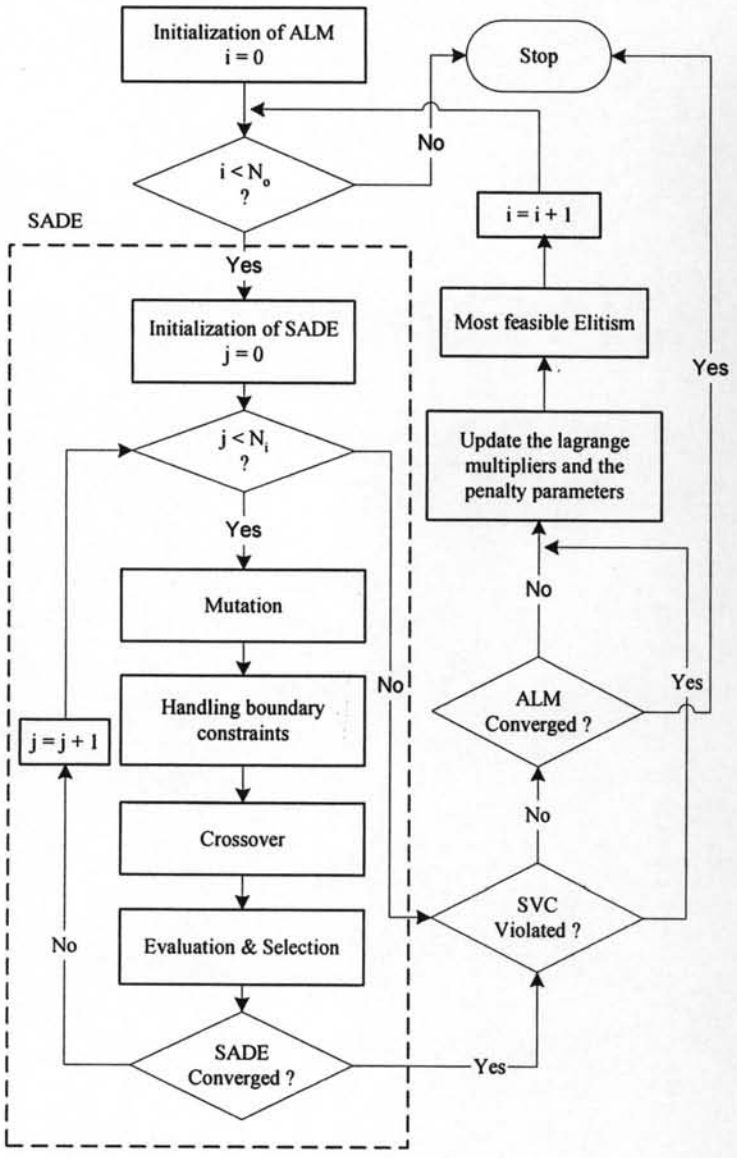

Figure 2.5 Flow chart of SADE_ALM (N_o : maximum number of iterations of outer loop, N_i : maximum number of iterations of inner loop).

$$x_{ij} = x_{ij,low} + \rho_{ij} \times \left(x_{ij,hi} - x_{ij,low}\right) \tag{2.30}$$

$$F_j = F_{j,low} + \rho_{1j} \times \left(F_{j,hi} - F_{j,low}\right) \tag{2.31}$$

$$CR_j = CR_{j,low} + \rho_{2j} \times \left(CR_{j,hi} - CR_{j,low}\right) \tag{2.32}$$

where F_j and CR_j are the mutation factor and the crossover constant for individual X_j, and ρ_{ij}, ρ_{1j}, and ρ_{2j} are uniformly distributed random numbers within [0,1] for individual x_{ij}, F_j, and CR_j respectively.

Step 2 (Mutation). For individual parent vector X_j, a mutant vector $X_j^{/}$ is created according to the following expression:

$$x'_{ij} = x_{ij,r_3} + F_j \times \left(x_{ij,r_1} - x_{ij,r_2}\right) \tag{2.33}$$

$$F'_j = F_{j,r_3} + F_j \times \left(F_{j,r_1} - F_{j,r_2}\right) \tag{2.34}$$

$$CR'_j = CR_{j,r_3} + F_j \times \left(CR_{j,r_1} - CR_{j,r_2}\right) \tag{2.35}$$

where r_1, r_2, and r_3 are randomly chosen indices such that r_1, r_2, and $r_3 \in (1,NP)$ and $r_1 \neq r_2 \neq r_3 \neq j$

Step 3 (Handling boundary constraints). In the event that mutation causes control variables, $x_{ij}^{/}$, $F_j^{/}$, and $CR_j^{/}$, exceeded their boundary constraints, i.e. lower or upper limit, such variables will be set to the nearest boundary.

Step 4 (Crossover). To increase the diversity of the mutant vectors, crossover is introduced to create the trial vector $X_j^{//}$ based on a series of n-1 binomial experiments [17] as shown below:

$$x''_{ij} = \begin{cases} x'_{ij}, \forall \rho_{ij} \leq CR_j \ \ or \ \ i = i_{rand} \\ x_{ij}, otherwise \end{cases} \tag{2.36}$$

$$F''_j = \begin{cases} F'_j, \forall \rho_{1j} \leq CR_j \ \ or \ \ i_{rand} = 1 \\ F_j, otherwise \end{cases} \tag{2.37}$$

$$CR''_j = \begin{cases} CR'_j, \forall \rho_{2j} \leq CR_j \ \ or \ \ i_{rand} = 2 \\ CR_j, otherwise \end{cases} \tag{2.38}$$

where ρ_{ij}, ρ_{1j}, and ρ_{2j} are the uniformly distributed random number within [0,1] for individual $x_{ij}^{//}$, $F_j^{//}$, and $CR_j^{//}$ respectively, and $i_{rand} \in (1,n+2)$ is a generated random integer to ensure that the trial vector $X_j^{//}$ is different from its associated parent vector X_j.

Step 5 (Evaluation and Selection). To create the new population in the next generation G+1, the fitness value, i.e. the augmented objective function value in (2.25) of the trial vector $X_j^{//(G)}$ is compared with its parent vector $X_j^{(G)}$ in the same way as in the classical DE as shown below:

$$X_j^{(G+1)} = \begin{cases} X_j^{''(G)}, \ if \ L_a\left(X_j^{''(G)}\right) \leq L_a\left(X_j^{(G)}\right) \\ X_j^{(G)}, \ otherwise \end{cases} \tag{2.39}$$

Step 6 (SADE termination criteria). The inner loop will be terminated according to two

criteria, i.e. 1) the maximum iteration of the inner loop (N_i); and 2) the convergence of the optimal solution as follows:

$$\Delta X_{opt} \le \varepsilon_{\Delta X} \tag{2.40}$$

where $\varepsilon_{\Delta X}$ is convergence tolerance of ΔX_{opt} which is determined by:

$$\Delta X_{opt} = \left\| X_{opt}^{(G)} - X_{opt}^{(G-1)} \right\|_{\infty} \tag{2.41}$$

where $\|.\|_{\infty}$ is the infinity-norm, $X_{opt}^{(G)}$ and $X_{opt}^{(G-1)}$ are the optimal solution obtained at current generation (G) and previous generation (G-1) respectively.

2.6.2 Outer loop iteration

After the inner loop has converged, the outer loop is started using the ALM algorithm to update the lagrange multipliers and the penalty parameters. The detail of the outer loop is described below.

Step 1 (Initialization of ALM). Set maximum iteration of the outer loop (N_o), the constrain violation tolerance (ε_{SVC}), the lagrange multiplier βs, and the penalty parameters r_g including c_g, and $r_{g.max}$.

Step 2 (Verifying constrain violation). The optimal solution obtained from the inner loop (X_{opt}^{*}) is checked that it is the feasible or infeasible solution through the sum of the violated constraints (SVC) index as follows:

$$SVC \le \varepsilon_{SVC} \tag{2.42}$$

where

$$SVC = \sum_{j=1}^{m} \left\{ max \left[g_j \left(X_{opt}^{*} \right), 0 \right] \right\} \tag{2.43}$$

If the optimal solution is violated go to step 4, else go to step 3.

Step 3 (Update the lagrange multiplier and the penalty parameters). The lagrange multiplier and the penalty parameters of the new augmented lagrange function La are updated according to (2.27) and (2.29) respectively before starting the next inner loop.

Step 4 (Applying the most feasible elitism). To improve the efficiency of the proposed algorithm, the most feasible elitism (X_{elite}) is employed by replacing the worst individual X_j which has the highest fitness value for the next inner loop iteration. The elitist member is

initialized by using the optimal solution obtained from the first inner loop iteration. Then, it is updated according to the extent of the violated *SVC* value and the objective function value f(·) (e.g., the total generator fuel cost which will be described in the following chapter) as follows:

1) If $SVC\left(X_{elite}^{(K-1)}\right) > \varepsilon_{SVC}$, then

$$X_{elite}^{(K)} = \begin{cases} X_{opt}^{*(K)}, \ if \ SVC\left(X_{opt}^{*(K)}\right) \le SVC\left(X_{elite}^{(K-1)}\right) \\ \qquad and \ f\left(X_{opt}^{*(K)}\right) \le f\left(X_{elite}^{(K-1)}\right) \\ X_{elite}^{(K-1)}, \ otherwise \end{cases} \tag{2.44}$$

2) If $SVC\left(X_{elite}^{(K-1)}\right) \le \varepsilon_{SVC}$, then

$$X_{elite}^{(K)} = \begin{cases} X_{opt}^{*(K)}, \ if \ f\left(X_{opt}^{*(K)}\right) \le f\left(X_{elite}^{(K-1)}\right) \\ X_{elite}^{(K-1)}, \ otherwise \end{cases} \tag{2.45}$$

where $X_{elite}^{(K)}$ and $X_{elite}^{(K-1)}$ are the elitist members of the current (K) and previous (K-1) iteration of the outer loop respectively, and $X_{opt}^{*(K)}$ is the optimal solution obtained from the current (K) iteration of the inner loop.

The outer loop will be terminated according to the same criteria as defined for the inner loop, i.e. 1) maximum iteration number of the outer loop (N_o), and 2) convergence of the optimal solution.

In the following chapter, we will introduce the problem of optimal economic operation of electric power system. The mathematical formulation of the economic dispatch problem and several variations on the representation of fuel-cost characteristics of thermal generating units are described. In addition, numerical examples of SADE_ALM described in this chapter are also provided in the next chapter.