# CHAPTER III

## SIZING OPTIMIZATION OF PLANAR STEEL TRUSSES

Several sizing optimization techniques are available in the context of truss design optimization. Most of the techniques are based on a common principle: to repeat the analysis process many times. Thus, the optimization process is normally time consuming, particularly if the nonlinearity effects are taken into account.

Most of the mathematical and numerical methods for optimization rely upon the assumption of continuity on both the design variables and the objective function. Under these assumptions, if the structural problem is actually discrete in nature, the resulting optimum values of the continuous design variables must be converted to appropriate discrete values. A conservative approach is to round to the larger values and to check that the constraints are still satisfied. Most of the continuous optimization techniques are gradient-based or deterministic. More recently, a number of probabilistic approaches such as genetic algorithms (GAs) and simulated annealing have been developed. A potential advantage of these methods is their inherent ability to accommodate discrete design variables and that they are free from limitations on the search space, e.g. continuity, differentiability and unimodality (Turkkan 2003).

The current study adopts the GAs for sizing optimization of the planar steel trusses. The objective function is the total self weight of the truss members (for the same kind of steel, it is equivalent to the total volume). The design variables – the member self weight (volume) – are in accordance with the practical section table in the design specification, resulting in a discrete problem. The constraints are converted to equivalent penalty terms.

### III.1. Genetic algorithms

Compared to traditional search and optimization procedures, such as calculus-based and enumerative strategies, the GAs are robust, global and generally more straightforward to apply in situations where there is little or no prior knowledge on the problem. As GAs require no derivative information or formal initial estimates of the solution, and because they are stochastic in nature, GAs are capable of searching the entire solution space with more likelihood of finding the global optimum.

Figure 3.1 illustrates the process in which a problem is solved by genetic algorithm, e.g. encoding the solutions, defining an objective function, using the genetic operators, etc., before performing the genetic search. During the search stage, the process is looped by many generations, using the fitness function to evaluate the possible optimum solution. At each generation, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation.

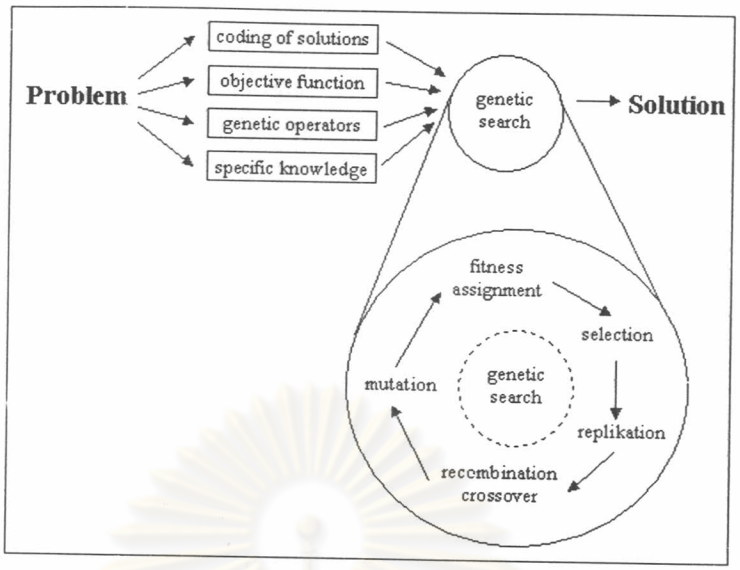


Figure 3.1 The genetic algorithm (Pohlheim 1997)

The fittest members of the initial population are given better chances of reproducing and transmitting part of their genetic heritage to the next generation. A new population is then created by recombination of the parental genes. It is expected that some members of this new population will have acquired the best characteristics of both parents and, being better adapted to the environmental conditions, will provide an improved solution to the problem.

After replacing the original population, the new group is submitted to the same evaluation procedure, and later generates its own offsprings. The process is repeated many times, until all members of a given generation share the same genetic heritage in which there are virtually no differences between individuals.

The members of these final generations, who are often quite different from their ancestors, possess genetic information that corresponds to the best solution to the optimization problem (Holland 1975).

*III.1.1 Definitions*

The terms used in genetic algorithms are borrowed to a large extent from natural genetics. Usually, genetic algorithms do not operate on the solutions alone but on a mapping of the solution that facilitates the application of the genetic operators. The set of all solutions in the problem space that can be represented by the genetic algorithms is called the *Phenotype*. The phenotype is mapped into a genetic space, known as the *Genotype*, where the genetic operators are carried out on the individuals. The mapping of the Phenotype to the Genotype is called *Genetic Coding*. Each individual in the genotype is called a *Chromosome*. A chromosome in turn is composed of *Genes* or *Alleles*. A particular genetic characteristic is governed by the value of the allele and its position in the chromosome. The allele's position in the chromosome is designated as its *Locus*.

The *Fitness* of an individual is a measure of its ability to survive and reproduce. The operator that allocates individuals of the present generation to the next

generation based on their fitness is called *Selection*. The operator that randomly combines two of the selected individuals is called *Crossover* and the operator that randomly changes the structure of an individual is the *Mutation* operator.

*III.1.2 GA basic parameters*

The basic parameters of GAs include population size, probability and type of crossover, and probability and type of mutation. By varying these parameters, the convergence of the problem may be altered. Thus, to maintain the robustness of the algorithm, it is important to assign appropriate values for these parameters (Pezeshk and Camp 2003). Much attention has been focused on finding the theoretical relationship among these parameters. Schwefel (1981) has developed theoretical models for optimal mutation rates with respect to convergence and convergence rates in the context of function optimization. De Jong and Spears (1990) have presented theoretical and empirical results on the interacting roles of population size and crossover in genetic algorithms. Cvetkovic and Muhlenbein (1994) have investigated the optimal population size for uniform crossover and truncation selection.

For a population size of 30-50, a probability of crossover $P_c$ of about 0.6 and a probability of mutation $P_m$ less than 0.01 is typical. The initial population, which might have been very far from the satisfactory solution, can adapt itself towards the optimized solution. Conversely, mutation tends to disorganize the convergence of the problem; therefore, the mutation rate, in conjunction with the population size, is crucial to the overall performance of GAs (Pezeshk and Camp 2003).

*III.1.3 Coding and Decoding*

An essential characteristic of GAs is the coding of the variables that describe the nature of the problem. For a specific problem that depends upon more than one variable, the coding is constructed by concatenating as many single variable codings as the number of the variables in the problem. The length of the coded representation of a variable corresponds to its range and precision. By decoding the individuals of the initial population, the solution for each specific instance is determined and the value of the objective function that corresponds to this individual is evaluated. This applies to all members of the population. There are many coding methods available, e.g. binary, gray, non-binary, etc. (Jenkins 1991a; 1991b; Hajela 1992; and Reeves 1993). The most common coding method is to transform the variables to a binary string of specific length.

According to Hajela (1992), an $r$-digit binary number representation of a continuous variable allows for $2^r$ distinct variations of that design variable to be considered. If a design variable is required to a precision of $\varepsilon$, then the number of digits in the binary string may be estimated from the following relationship:

$$2^r \geq \frac{(X_u - X_l)}{1+\varepsilon} \tag{3.1}$$

where $X_u$ and $X_l$ are the upper and lower bounds of a continuous variable $X$.

Hajela (1992) has suggested that although a higher degree of precision may be obtained by increasing the string length, higher degree of schema disruption can be expected. In addition, larger defined length schema clearly are disadvantageous in dominating the population pool. For example, a real variable $X$ in the range $0.0<X<5.0$ can be coded as a 3-digit string: $000 \leq X \leq 111$.

There are a total of $2^3 = 8$ points (values) in this range. Of the $2^r$ possible $r$-digit binary strings, a unique string is assigned to each of the $n$ integer variables. In this example, there are six integers between 0 and 5; therefore, there are $2^3$ binary strings. Hajela (1992) has recommended that two extra binary values be assigned to the out-of-bound variables 6 and 7: [0, 1, 2, 3, 4, 5, 6*, 7*] $\Leftrightarrow$ [000, 001, 010, 011, 100, 101, 110*, 111*], whereas * indicates an out-of-bound variable. A penalty measure can then be applied to the fitness function of a design variable that includes the out-of-bound integer variable.

Another approach is a one-to-one correspondence between the integer variables and their binary representation. Decoding from a binary number to a real number can be performed using the following equation (Adeli and Cheng 1994):

$$C = C_{\min} + \frac{B(C_{\max} - C_{\min})}{2^L} \qquad (3.2)$$

where $C$ is the real value of the string;
$C_{min}$ and $C_{max}$ are the lower and upper bounds of $C$;
$L$ is the length of the binary string; and
$B$ is the decimal integer value of the binary string.

*III.1.4 Selection*

The selection operator is intended to improve the average quality of the population by giving individuals of higher fitness a higher probability to be copied into the next generation.

The roulette wheel selection is the most basic selection method. For example, let us consider the generation (population) of 10 individuals (chromosomes) with the fitness values shown in the table in Figure 3.2.

| Individuals | Fitness |
|---|---|
| 1 | 0.18 |
| 2 | 0.16 |
| 3 | 0.15 |
| 4 | 0.13 |
| 5 | 0.11 |
| 6 | 0.09 |
| 7 | 0.07 |
| 8 | 0.06 |
| 9 | 0.03 |
| 10 | 0.02 |
| Total | 1.00 |

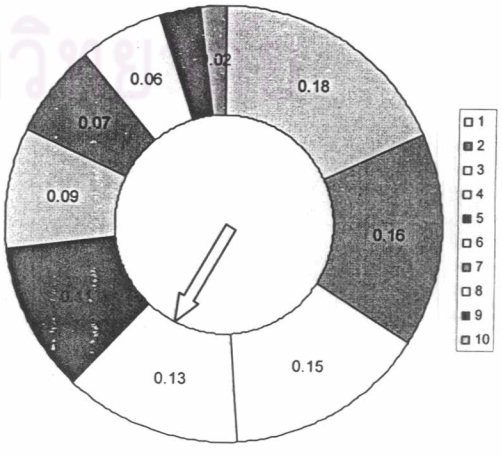


Figure 3.2 The roulette wheel.

Note that the sum of all fitness values is equal to 100%. These values will be arranged on the roulette wheel; the greater its value, the larger its angle. Each time the wheel runs, an individual is selected; the higher its fitness, the higher probability it can be selected. In practice, all individuals are lined up one by one. Instead of the wheel, a random number varying from zero to one will be generated to select the individual.

The basic roulette wheel selection has a limitation in that for many deceptive problems, it tends to converge to certain sub-optimal regions of the search space, because it favors only above average chromosomes of a particular generation. Hence, a below average chromosome, which on crossover or mutation could have given a fitter chromosome is not selected, and the solution converges prematurely to a sub-optimal region. Hence, to maintain adequate diversity in the population, some other selection schemes have been implemented, e.g. proportional selection, fitness-scaling selection, group selection, etc. (Pezeshk and Camp 2003).

*III.1.5 Crossover*

The crossover operator is intended to combine the genetic data of the existing population and the generated offsprings. A pair of chromosomes is recombined on a random basis to form two new individuals. There are many types of crossover, such as one-point crossover (standard), multi-point crossover, uniform crossover, adapting crossover, etc. (Pezeshk and Camp 2003).

The standard one-point crossover has been shown to be deficient in the optimization of deceptive problems and hence other crossover schemes, like a multi-point crossover, are often performed. Besides, a uniform crossover tends to create diversity and hence favors exploration of the search space. However, this usually tends to hinder convergence and is not used except in very special cases like massively multimodal domains. Figure 3.3 illustrates the one-point and multi-point crossover schemes.

Two individuals A and B:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A = | *0* | *1* | *0* | *1* | *1* | *1* | *0* | *1* | *1* | *0* | *0* | *1* |
| B = | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

One-point crossover (at k = 6):

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A' = | *0* | *1* | *0* | *1* | *1* | *1* | 1 | 1 | 0 | 1 | 1 | 0 |
| B' = | 1 | 0 | 1 | 1 | 0 | 0 | *0* | *1* | *1* | *0* | *0* | *1* |

Multi-point crossover (at k = 1, 4, 7, 10):

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A' = | *0* | 0 | 1 | 1 | *1* | *1* | *0* | 1 | 0 | 1 | *0* | *1* |
| B' = | 1 | *1* | *0* | *1* | 0 | 0 | 1 | *1* | *1* | *0* | 1 | 0 |

Figure 3.3 An example of crossover.

One obvious way for the GA to self-adapt its use of different crossover operators is to append two bits to the end of every individual in the population. Suppose "00" refers to a one-point crossover, "01" to a two-point crossover, "10" to a three-point crossover, and "11" to a uniform crossover. Then, the last two columns of the population are used to sample the crossover operator space. If the uniform crossover moves the search into the solution space with high fitness, then more 11's should appear in the last two columns as the GA evolves. If higher fitness solutions are found using the two-point crossover, more 01's should appear accordingly. Because the approach is self-adaptive, crossover and mutation are allowed to manipulate these extra two columns of bits (Pezeshk and Camp 2003).

*III.1.6 Recombination*

Crossover in fact is a special branch of recombination for binary numbers. For real numbers, the process is called recombination. There are many approaches available in the literature such as discrete recombination, intermediate recombination, line recombination, and extended line recombination (Pohlheim 1997).

Discrete recombination performs an exchange of variable between the individuals. It generates corners of the hypercube defined by the parents. It can be used with any kind of variables (binary, real or symbols). The schematic representation of recombination is illustrated in Figure 3.4.

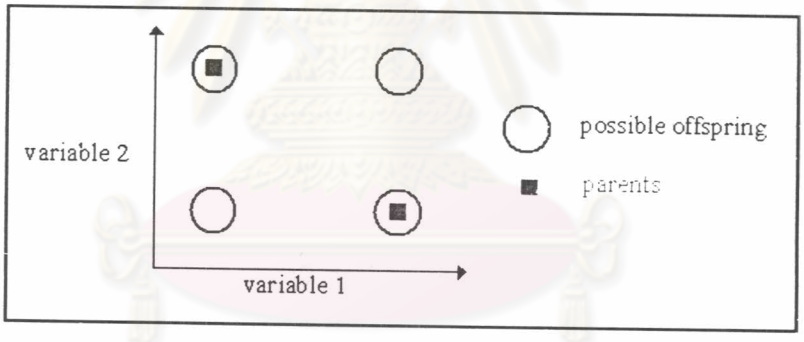


Figure 3.4 Possible positions after discrete recombination (Pohlheim 1997)

Most recombination approaches are The last three approaches based on the same rule of offspring production, that is

$$\text{Offspring} = \text{Parent 1} + \alpha\,(\text{Parent 2} - \text{Parent 1}) \qquad (3.3)$$

where $\alpha$ is a scaling factor chosen randomly within an interval $[-d, 1 + d]$ in which $d = 0$ and $d > 0$ for intermediate and extended intermediate recombination, respectively. Line recombination is similar to intermediate recombination, except that only one value of $\alpha$ for all variables is used. An optimum value of $\alpha$ might be equal to 0.25 as shown in Figure 3.5

Intermediate recombination is capable of producing any point within a hypercube slightly larger than that defined by the parents. Line recombination can generate any point on the line defined by the parents. Extended line recombination tests more often outside the area defined by the parents and in the direction of parent 1. The probability of small step sizes is greater than that of bigger steps.

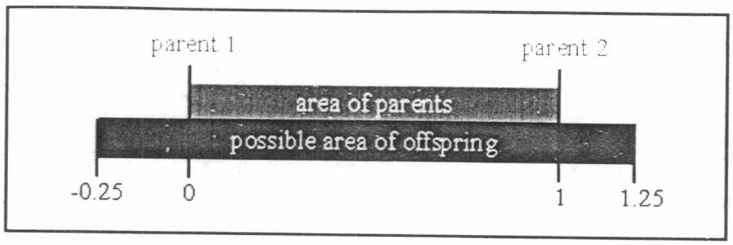


Figure 3.5 Area for variable value of offspring (Pohlheim 1997)

*III.1.7 Mutation*

The mutation operator is intended to make a small change in the characteristics of an individual (chromosome). It allows new genetic patterns to be created, thus improving the search results. Occasionally, it protects some useful genetic material loss. During the process, a rate of mutation determines the possibility of mutating one of the design variables. Usually this probability is kept very low, typically around 0.001 to create sufficient diversity but at the same time not to hinder convergence.

When mutation is applied to a string, it sweeps down the string of bits, and changes the bit from 0 to 1 or from 1 to 0 if a probability test is passed. Mutation has an associated probability parameter that is typically quite low. By definition, mutation is a random walk through the string space.

An example below (Figure 3.6) shows two parent chromosomes of length 5 with randomly generated numbers used for the mutation probability check (0.002), and the resulting mutated chromosomes. It is observed that for the first chromosome, the probability test is never passed, and the output of the mutation is the same as the input. In the second case, the probability test is passed for the second bit. Thus, this bit is changed from 0 to 1.

| Before | | | | | Probability test | | | | | After | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0.753 | 0.659 | 0.600 | 0.035 | 0.725 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0.646 | 0.001 | 0.990 | 0.859 | 0.394 | 1 | 1 | 1 | 1 | 0 |

Figure 3.6 An example of mutation.

Like recombination and crossover, there is real-value mutation in addition to binary mutation. The real-value mutation algorithm generates most points in the hypercube defined by the variables of the individual and the range of the mutation. The size of the mutation step is usually difficult to choose. The optimal step size depends upon the problem and may even vary during the optimization process. Small steps are often successful, but sometimes bigger steps are quicker. Figure 3.7 schematically illustrates the effect of mutation.

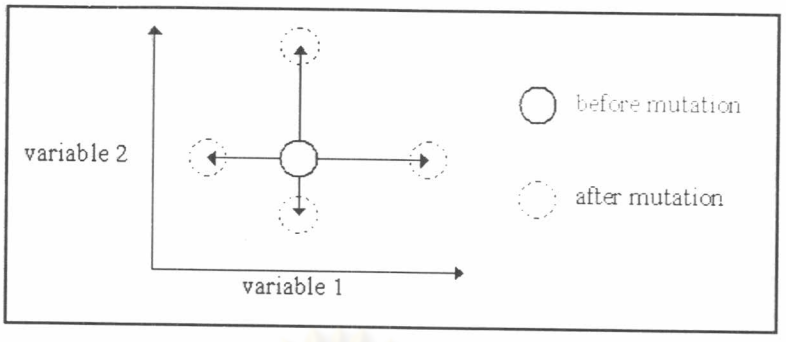


Figure 3.7 Effect of mutation (Pohlheim 1997)

*III.1.8 Penalty functions*

To evaluate the performance or fitness of a particular solution string, the string's characters are decoded into series of the design variables. Using these design variables, an analysis is performed and the objective function is computed. If any constraints are violated, a penalty term is applied to the objective function, with the value of the penalty related to the degree in which the constraints are violated. The resulting penalized objective function quantitatively represents the extent of the violation of the constraints and provides a relatively meaningful measurement of the performance of each solution string. Several penalty function schemes have been proposed for structural design (Pezeshk and Camp 2003).

One of the simplest penalty functions is a multiple linear segment function. Consider a problem where the displacement and stress constraints are imposed. Each of the structural member is checked for stress violation, and each node is checked for displacement violation. If no violation is found, then no penalty term is imposed on the objective function. If a constraint is violated, then the penalty term is defined as (Pezeshk and Camp 2003):

$$\theta_i = \begin{cases} 1 & if \ \dfrac{|P_i|}{P_{\max}} \le 1 \\ \dfrac{k_1 |P_i|}{P_{\max}} & if \ \dfrac{|P_i|}{P_{\max}} > 1 \end{cases} \qquad (3.4)$$

where $\theta_i$ is a penalty value for constraint $i$;
$P_i$ is a structural parameter or response (deflection, stress, etc.);
$P_{max}$ is the maximum allowable value of each $p_i$; and
$k_1$ is the penalty rate.

Another type of the penalty terms is a nonlinear function which is defined as (Pezeshk and Camp 2003):

$$\Phi_i = 1 + k_2 (q_i - 1)^n \qquad (3.5)$$

where $k_2$ is the nonlinear penalty rate; $n$ is the order of nonlinearity, and $q_i$ is similar defined as:

$$q_i = \begin{cases} 1 & if \dfrac{|P_i|}{P_{\max,i}} \le 1 \\ \dfrac{|P_i|}{P_{\max,i}} & if \dfrac{|P_i|}{P_{\max,i}} > 1 \end{cases} \tag{3.6}$$

Typical penalty functions are shown in Figure 3.8.

Having obtained the penalty function factors, the fitness value of a particular solution string is obtained by multiplying the objective function (structural weight) by the corresponding penalty factors:

$$F = W \prod_{i=1}^{n} \theta_i \tag{3.7}$$

where $F$ is the string fitness (penalized objective function);
$n$ is the total number of points where the constraints are checked;
$W$ is the weight (volume) of the entire structure (objective function), and the product represents the total penalty.

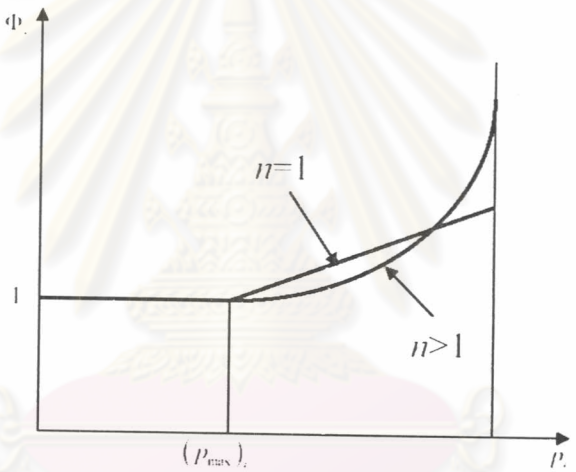


Figure 3.8 Typical penalty functions (Pezeshk and Camp 2003).

## III.2. Sizing optimization of planar steel trusses

Let us consider an ideally-pinned planar truss structure, subjected to static point loads and self weight. By assuming that all of the truss members have the same material density, the optimization problem may be expressed as (Pezeshk and Camp 2003):

$$\text{Minimize} \qquad W = \sum_{i=1}^{m} A_i L_i \tag{3.8}$$

$$\text{Subject to} \qquad G = \begin{cases} s^l \le s \le s^u \\ d^l \le d \le d^u \\ A^l \le A \le A^u \end{cases} \tag{3.9}$$

where $i$ is the index of the truss member, varying from 1 to the total number of members $m$.
$L_i$ is the length of member $i$.

$A_i = A_i(\eta)$ is the cross-sectional area of member $i$, with $\eta$ representing the reference number for a given section.

In the above equation, the vectors s, d, A contain the values of stresses, displacements, and cross-sectional areas, respectively. The superscripts l and u refer to the prescribed lower and upper bounds of each constraint.

*III.2.1 Algorithm*

The algorithm begins by creating a random initial population, or the first generation. The population size is the number of solutions considered in each generation.

The next step is to create a sequence of new populations, or generations. At each step, the individuals in the current generation are the source to create the next generation.

To create a new generation, the algorithm performs the following steps: scoring each member of the current population by computing its fitness value; scaling the raw fitness scores to convert them into a more usable range of values; selecting parents based on their fitness (selection operator); and producing children from the parents (crossover operator). The children are produced either by making random changes to a single parent (mutation) or by combining the vector entries of a pair of parents (crossover).

Subsequently, the current population is replaced with the children to form the next generation. The algorithm stops when one of the termination criteria is met. The termination criteria are the number of generations to be created, the total computation time, the number of continuous generations without any significant improvement, and the running time without any significant improvement (MATLAB® 2004).

The flowchart in Figure 3.9 demonstrates the main algorithm.

*III.2.2. MATLAB GAs Toolbox*

MATLAB® is a high-performance language for technical computing that integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. This is an interactive environment whose basic data element is an array that does not require dimensioning. The greatest advantage of MATLAB is its capability to solve many mathematical problems, particularly those with matrix and vector formulations, similarly with the scalar in other programming languages. However, MATLAB® is a translating language which is a huge drawback due to its slow speed in computation.

MATLAB features a family of application-specific solutions called toolboxes. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. All the toolbox functions are MATLAB M-files, made up of MATLAB statements that implement specialized optimization algorithms. The Genetic Algorithm and Direct

Search toolbox is a collection of functions that extend the capabilities of the Optimization toolbox and the MATLAB numeric computing environment.

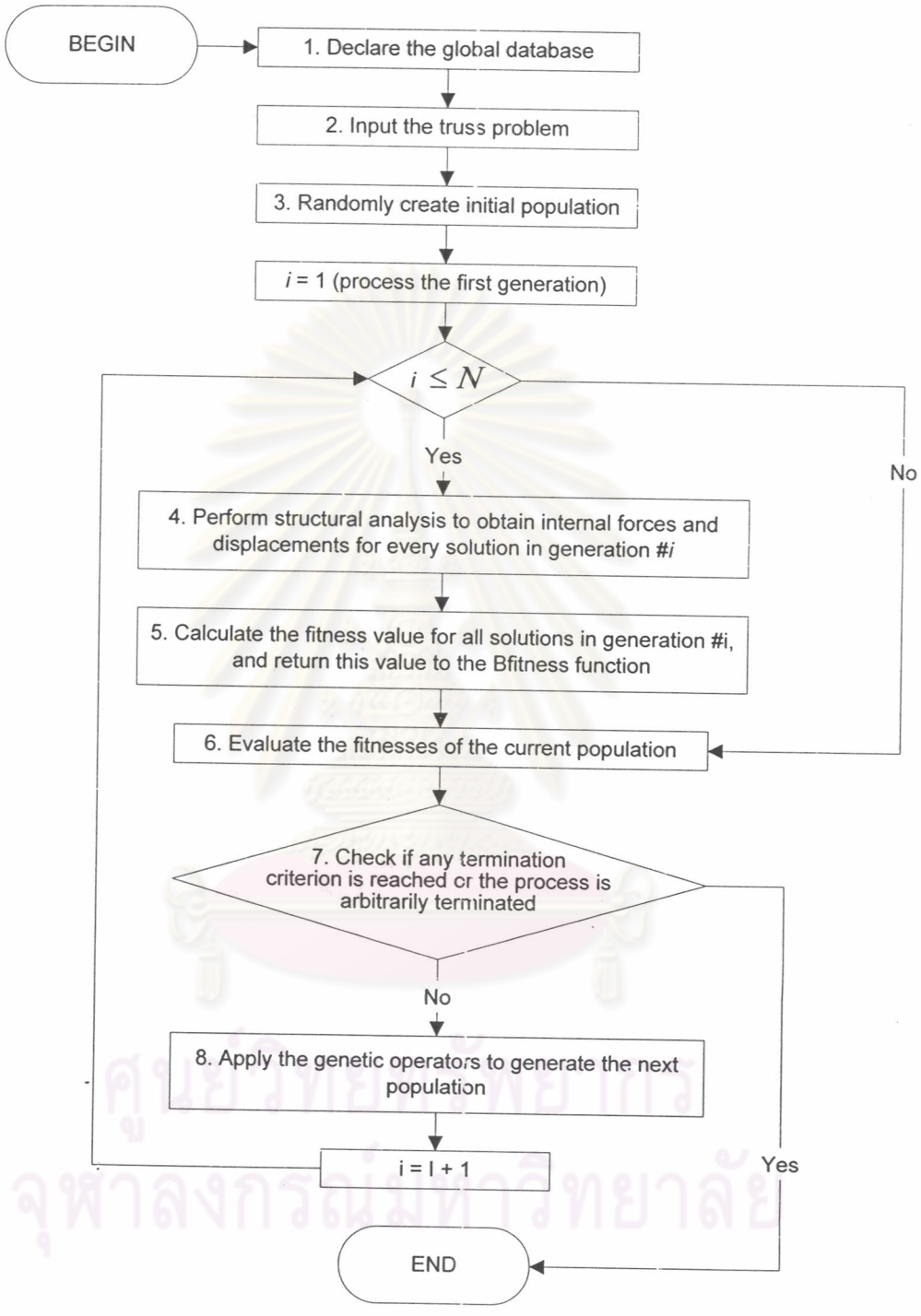


Figure 3.9 The main flowchart of the genetic algorithm

The current works extend the capabilities of the GAs toolbox by writing its own M-files, to utilize the toolbox in the stage of GA optimization. By default, the GA function in MATLAB finds the minimum value of the objective function for a specified problem.

*III.2.3. Penalized objective function*

There are three types of constraints in this problem: the allowable stresses, the allowable displacements, and the available cross-sectional areas.

The variety in the cross-sectional areas is accounted for by using an input list of available sections. Because the selection is based upon this list, we can change the number of sections, the order and the value of each section conveniently.

For the displacement constraint, the penalty term is computed as follows:

$$\theta_{1i} = \begin{cases} 1 & if \ \dfrac{|P_i|}{P_{\max}} \le 1 \\ \dfrac{k_1 |P_i|}{P_{\max}} & if \ \dfrac{|P_i|}{P_{\max}} > 1 \end{cases} \tag{3.10}$$

$$\theta_1 = \prod_{i=1}^{n} \theta_{1i} \tag{3.11}$$

where $|P_i|$ represents the displacement of the degree of freedom $i$ and $P_{\max}$ denotes the maximum allowable displacement.

For the constraints upon stress in linear analysis, the penalty term can be computed similarly as for the case of the displacement constraints. However, tracing the loading history in nonlinear analysis is viewed a better way to determine the penalty term for stress constraints.

Let us denote $\lambda_{initial}$ as the load ratio at which the first member of the truss fails (yielding or buckling); $\lambda_{\max}$ as the load ratio at which the truss reaches its maximum deflection; $\lambda_{critical}$ as the load ratio at which the last member of the truss fails, i.e. the truss is about to collapse; and $\lambda_k$ as a specified load ratio varying from $\lambda_{initial}$ to $\lambda_{\max}$ or $\lambda_{critical}$.

The case in which $\lambda > 1.0$ means the truss can be subjected to heavier load compare to the current load. With the same amount of truss (steel) volume, a set of cross-sectional areas for the members which has a higher value of $\lambda$ means a better solution. However, this also means that the truss is not an optimum design. The key objective is to minimize the volume of the truss, with the lowest value of $\lambda$ slightly greater than 1.0. The upper bound of $\lambda$ is thus set as $\lambda = 5$. On the other hand, $\lambda < 1.0$ means the truss is overloaded. To increase the slope on this side, the penalty function is set as $\theta_2 = 5$ at $\lambda = 0$.

Based upon the above discussion, $\theta_2$ should be chosen such that its lowest value is 1.0 (the optimum $\lambda$). The range $0.0 < \lambda \le 1.0$ is avoided by using a parabolic

function of $\theta_2$ as well as the range $1.0 < \lambda \leq 3.0$ but with a more gradual slope ($\theta_{2i} \to 3.0$ as $\lambda \to 3.0$).

The penalty function for stresses is computed as follow:

$$\theta_{2i} = \begin{cases} 4(\lambda_i - 1)^2 + 1 & if\ 0.0 < \lambda_i \leq 1.0 \\ \dfrac{(\lambda_i - 1)^2}{2} + 1 & if\ 1.0 < \lambda_i \leq 3.0 \\ 3.0 & if \quad 3.0 < \lambda_i \end{cases} \tag{3.12}$$

$$\theta_2 = \prod_{i=1}^{n} \theta_{2i} \tag{3.13}$$

Figure 3.10 graphically summarizes $\theta_{2i}$.

The penalized objective function of a particular solution string is obtained by multiplying the truss volume with the corresponding penalty factors:

$$F = \theta_1 \theta_2 V \tag{3.14}$$

where $F$ is the fitness value (penalized objective function); and $V$ is the volume of the truss under consideration.

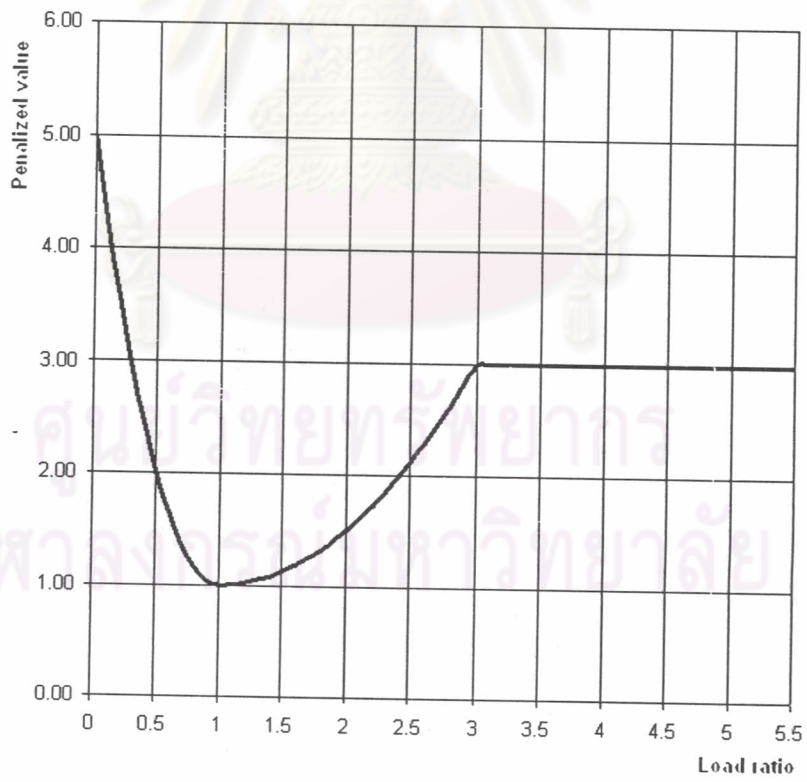


Figure 3.10 The penalty function