

SIMPLE GENETIC ALGORITHM WITH SIMPLEX LOCAL
TUNING FOR EFFICIENT GLOBAL
OPTIMIZATION

By

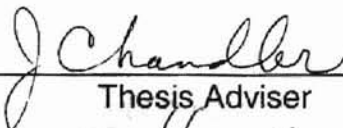
JIANPING LU

Bachelor of Science
Changchun University
Science and Technology
Changchun, China
1982

Submitted to the Faculty of the Graduate College
Of the Oklahoma State University
In partial fulfillment of the requirements
For the Degree of
MASTER OF SCIENCE
May, 2000

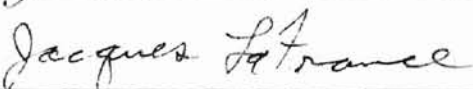
SIMPLE GENETIC ALGORITHM WITH SIMPLEX LOCAL
TUNING FOR EFFICIENT GLOBAL
OPTIMIZATION

Thesis Approved:



Thesis Adviser







Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to acknowledge, with deepest appreciation, a list of people who contributed their interests, efforts, and supports for the completion of this document. I must first thank my chairperson, J. P. Chandler, for his guidance, patience, and support. His unwavering commitment to my academic growth, extraordinary professional expertise, intellectual knowledge, and purity of moral character provided me with a mentor of the highest standard. Secondly, I would like to thank my committee members, Dr. G. E. Hedrick and Dr. Jacques LaFrance, for their insight and thoughtful comments concerning my researches and for providing the valuable time to read and polish this document.

There is no doubt that I would not have accomplished this document without the incredible love and support from my family. My wife Wei Li, her understanding, love, kindness and encouragement gave me the strength to reach this goal. My daughter, Zhen Lu, and my son, Jeffery Lu, gave me so much memorable time during my completion of this document.

TABLES OF CONTENTS

Chapter	Page
I. INTRODUCTION.....	1
1.1 The Principle of Genetic Algorithms.....	1
1.2 Simplex Optimization.....	6
1.3 Artificial Neural Network.....	10
II. METHODOLOGY.....	15
2.1 Research Background.....	15
2.2 Combination with the Simplex Downhill Method.....	15
2.3 Consideration of Genetic Diversity	17
2.4 Basics of Simple Genetic Algorithm with Simplex Local Tuning ...	18
IV. RESULTS AND DISCUSSION.....	23
3.1 Case 1.....	23
3.2 Case 2.....	25
3.3 Case 3.....	41
V. CONCLUSIONS.....	49
REFERENCES.....	50
APPENDIX A – PROGRAM LISTING.....	53

LIST OF TABLES

Table	Page
I. Comparison of the impact of using genetic diversity guidance for the simplified genetic algorithm.....	24
II. Training Set for the $K^+/Ca^{2+}/NO_3^-/Cl$ system.....	25
III. Generation data for over-fit.....	28
IV. Generation data for elimination of over-fit.....	31
V. Comparison of the pure genetic algorithm and the combination with downhill method.....	35
VI. Training result of data set in Table 2 using GD method.....	35
VII. Test data set for the $K^+/Ca^{2+}/NO_3^-/Cl$ system	37
VIII. Test result for data set in table 7 using GD method.....	38
IX. Sensitivities to the parameters of the proposed algorithm.....	40
X. Data for the lubricant viscosity at different temperatures and pressures..	42
XI. Performance of the proposed algorithm (GD) case 3.....	44
XII. Training data for the lubricant viscosity using proposed algorithm.....	44
XIII. Test and generalization data for the proposed algorithm case 3.....	47

CHAPTER
INTRODUCTION

1.1 The Principles of Genetic Algorithms

Figure	LIST OF FIGURES	Page
I.	Flowchart of a genetic algorithm.....	2
II.	One-point crossover examples.....	3
III.	Two-point crossover examples.....	4
IV.	Hypothetical population and its fitness.....	5
V.	Pie chart of fitness.....	6
VI.	Illustration of the relationship between the vertices for simplex optimization.....	8
VII.	Three-layer neural network architecture.....	12
VIII.	Composition of neurons.....	13
IX.	Over-fit phenomenon.....	30
X.	Elimination of over-fit.....	34

CHAPTER I

INTRODUCTION

1.1 The Principle of Genetic Algorithms

Genetic algorithms (GAs) are optimization techniques based on the concepts of natural selection and genetics [1-5]. In this approach, the variables are represented as genes on an artificial chromosome. Similar to Simplex optimization (this will be discussed in 1.2), GAs feature a group of candidate solutions (population) on the response surface. Through natural selection and the genetic operators, mutation and recombination, chromosomes with better fitness (response function scores) are found. Natural selection guarantees that chromosomes with the best fitness will propagate in future populations. Using the recombination operator, the GA combines genes from two parent chromosomes to form two new chromosomes (children) that have a high probability of having better fitness than their parents. Mutation allows new areas of the response surface to be explored. One of the reasons GAs work so well is that they offer a combination of hill-climbing ability (natural selection) and a stochastic method (recombination and mutation).

The simple GA is comprised of four steps as shown in the flowchart in Figure 1-1. First, the initial population of chromosomes is created either randomly or by perturbing an input chromosome. The population size (N_p) remains constant throughout the optimization and is a user-controlled option. In the second step, evaluation, the fitness is computed. The third step is the exploitation or natural selection step. In this step, the chromosomes with the

crossover, mutation and selection. The crossover, mutation, selection cycle of the GA is known as a generation, equivalent to the iterations of traditional

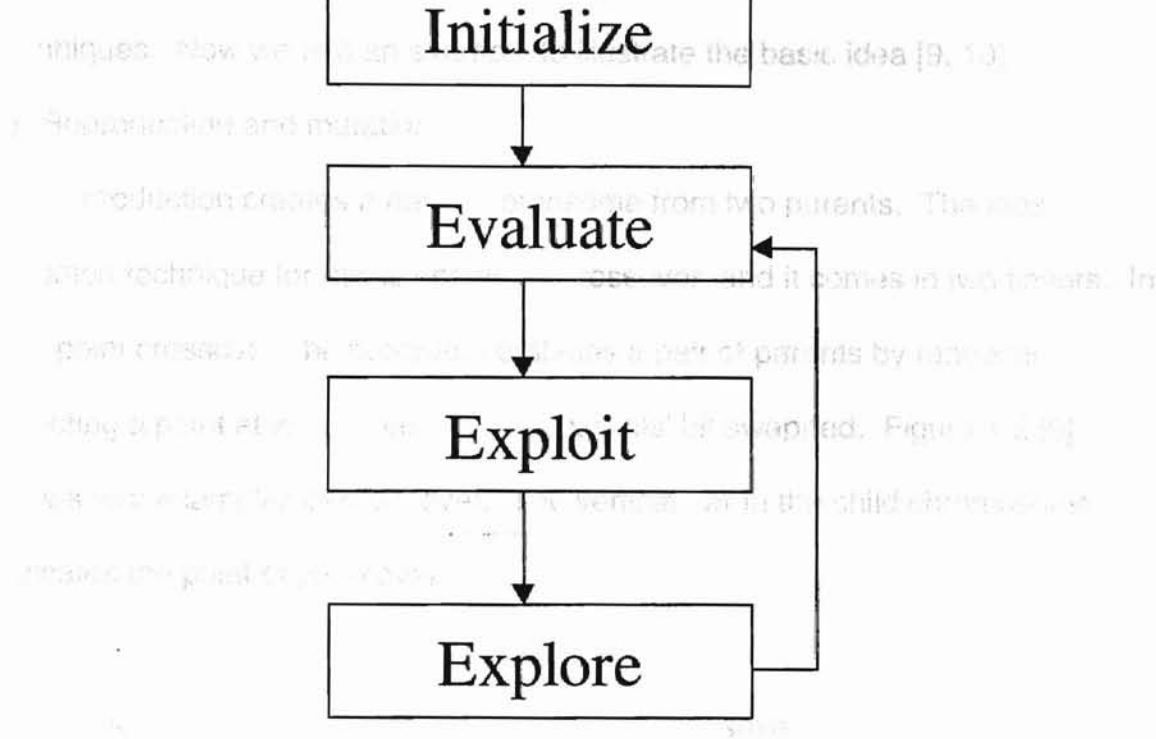


Figure 1-1 Flowchart of a genetic algorithm [7].

largest fitness score are placed one or more times into a mating subset in a semi-random fashion. Some chromosomes with low fitness scores are removed from the population. There are several methods for performing exploitation. One of the most common methods is the binary tournament mating subset selection method and is discussed in reference 8. The fourth step, exploration, consists of the recombination and mutation operators. Thus the principal data object of the GA is the chromosome and its utility is generally known as its fitness. The principal operators which manipulate these chromosomes are known as

crossover, mutation and selection. The crossover, mutation, selection cycle of the GA is known as a generation and is equivalent to the iterations of traditional techniques. Now we use an example to illustrate the basic idea [9, 10].

1) Reproduction and mutation

Reproduction creates a new chromosome from two parents. The most common technique for this is known as crossover, and it comes in two flavors. In one-point crossover, the program combines a pair of parents by randomly selecting a point at which pieces of the parents' bit swapped. Figure 1-2 [9] shows four examples of crossover. The vertical bar in the child chromosome indicates the point of crossover.

Father Chromosome	Mother Chromosome	Crossover Point	Child Chromosome
10010011	10110110	3	100 10110
10000000	10110110	6	100000 10
10110110	11101110	2	10 101110
10110110	11101110	5	10110 110

Figure 1-2. One-point crossover examples.

Another technique is two-point crossover, which swaps the beginning and end of one parent and the middle of another, using two randomly selected bit

positions. Figure 1-3 [9] shows two-point crossover works. Crossover allows the mixing of attributes from different chromosomes.

Father Chromosome	Mother Chromosome	Cross Point 1	Cross Point 2	Child Chromosome
10010011	10110110	3	6	100 1011 1
10000000	10110110	0	4	1000 0110
10110110	11101110	2	3	10 10 0110

Figure 1-3. Two-point crossover examples.

Reproduction also involves mutation, a random change of one or more bits in each chromosome of the new population. The primary purpose of mutation is to increase variation in a population; mutation is most important in populations where the initial population may be a small subset of all possible solutions.

2) The Selection Operator

The selection operator ensures that the number of representatives a chromosome receives in the following generation is dependant upon its fitness in proportion to the average fitness of the current population. The most common technique is Roulette Wheel Selection [11], a spinning circle divided into different pie-shaped slots. For a genetic algorithm, each slot on the wheel represents a chromosome from the parent generation; the width of each slot represents the

relative fitness of a given chromosome. Essentially, the simulated roulette wheel generates a random number that is some fraction of the total fitness of the parent population; then it counts around the wheel until it finds the selected chromosome. The largest fitness values tend to be chosen since they have larger slots.

Let's look at a small example with a hypothetical population of five. Figure 1-4 [9] shows a five-member population and its corresponding fitness values.

Order	Chromosome	Fitness
1	10110110	20
2	10000000	5
3	11101110	15
4	10010011	8
5	10100010	12

Figure 1-4. Hypothetical population and its fitness.

The total fitness of this population is 60. Figure 1-5 shows the relative size of pie slices as assigned by fitness. Chromosome 101101110 (order 1) has a 34% chance of being selected as a parent, where 10000000 (order 2) has only an 8% chance of generating a new chromosome. Each chromosome in a new generation will be parented by chromosomes selected, by fitness, from the old generation.

consisted of a series of two-level factorial designs with the concept that the

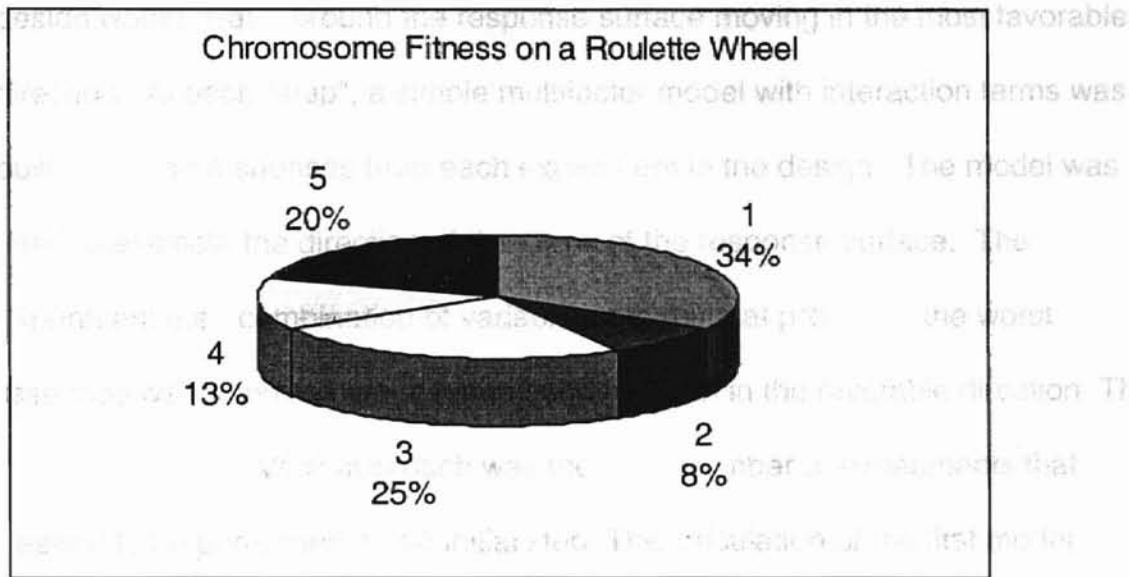


Figure 1-5. Pie chart of fitness.

The stopping condition is usually dependent either upon some fitness criterion having been reached or upon a certain number of generations having taken place.

1.2 Simplex Optimization

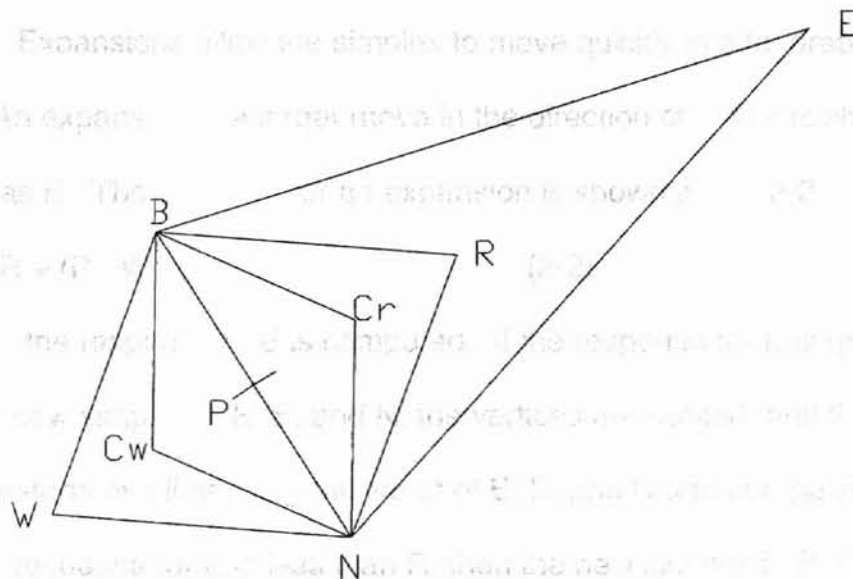
Simplex optimization methods combine response surface methodology such as experimental designs and hill-climbing approaches such as steepest ascent/descent. Box and coworkers developed the first simplex-type method and noted the similarities between this method and Darwin's theory of evolution, hence the name evolutionary optimization (EVOP) [12,13]. EVOP was able to vary multiple factors simultaneously and to make intelligent guesses as to what experiments should be performed next. Unlike experimental design methods, the variable settings to be studied were not known in advance. This method

consisted of a series of two-level factorial designs with the concept that the design would "walk" around the response surface moving in the most favorable direction. At each "step", a simple multifactor model with interaction terms was built using the responses from each experiment in the design. The model was used to estimate the direction of the slope of the response surface. The experiment (i.e., combination of variable settings) that produced the worst response was removed and a new step was taken in the favorable direction. The drawback to the EVOP approach was the large number of experiments that needed to be performed in the initial step. The calculation of the first model required 2^n experiments, where n is the number of variables being optimized.

In 1962, Spendley *et al.* developed a fixed-size sequential simplex method that was more efficient than the EVOP method [14]. A simplex is defined as a geometric figure consisting of one more vertex than the number of variables being optimized. By employing a simplex instead of a two-level factorial design, $n + 1$ versus 2^n experiments were required initially. Although the fixed-size simplex is able to reach the global optimum region, it has difficulties in finding the exact optimum due to its rigid shape.

In 1965, Nelder and Mead tried to improve the method of Spendley by giving the simplex the ability to accelerate in favorable directions, decelerate in poor directions, and change its shape [15]. Their algorithm, termed the variable-sized simplex, is very simple and consists of five logical steps. The possible movements for the variable-sized simplex in a two-dimensional case are illustrated in Figure 1-2. First, an initial simplex is created either randomly or by

perturbing a vector of input variable settings. The coordinates of the vertices in the simplex represent individual combinations of variable settings. Each vertex is ranked according to its corresponding response score.



In Figure 1-6, the vertices have been ranked and are designated B, N, and

Figure 1-6. Illustration of the relationships between the vertices for simplex optimization. P is the middle point between B and N. W for best, next-to-worst, and worst, respectively. The centroid, P, of the simplex is computed by averaging the coordinates for all vertices except the worst one. Next, a step is away from the worst vertex by reflecting through P as shown in Eq. 2-1.

$$R = P + (P - W) \quad (2-1)$$

Distance WP should equal PR. WC_w should equal C_wP and PC_p . This new step is shown in Figure 1-6 as R. The new simplex replaces the old simplex and now consists of the vertices B, N, and R. Based on the response score for R, one of three moves can be taken.

First, if the response score of R is greater than B, then an expansion is performed. Expansions allow the simplex to move quickly in a favorable direction. An expansion is a larger move in the direction of R and is shown in Figure 1-6 as E. The equation for an expansion is shown in Eq. 2-2.

$$E = R + (P - W) \quad (2-2)$$

Next, the response of E is computed. If the response for E is greater than R, then the new simplex is E, B, and N, the vertices are ranked, and the algorithm restarts by eliminating the worst of E, B, and N and computing a new step. If the response for E is less than R, then the new simplex is R, B, and N, the vertices are ranked, and the algorithm repeats as above.

The second possible case occurs if the response at R is less than the response at B, but greater than at N. In this case, neither expansion nor contraction is performed. The new simplex is B, R, and N, the vertices are ranked, and the algorithm restarts. The third case occurs if the response score at R is less than at N. In this case, a new vertex is selected with the span of the previous simplex. This is termed a contraction. Contractions allow the simplex to shrink in size. If the response at R is less than at N, but greater than at W, then a contraction closer toward R is made by use of Eq. 2-3.

$$C_r = P + 0.5(P - W) \quad (2-3)$$

The new simplex is B, N, and Cr, the vertices are ranked, and the algorithm restarts. If the response at R is less than at W, then a contraction closer to W is be taken by use of Eq. 2-4.

$$C_w = P - 0.5(P - W) \quad (2-4)$$

This is termed a negative contraction. The new simplex consists of B, N, and Cw, the vertices are ranked, and the algorithm restarts. Movements (iterations) are made until a termination criterion such as a fixed number of movements is met.

The primary disadvantage of Simplex optimization is the inability to move from local optima that may occur on the response surface. Furthermore, according to the investigation of Virginia Torczon [16], the simplex procedure of Nelder and Mead is inherently not robust and fails disastrously when the number of variables is as large as 16, and sometimes when it is as large as 8. Thus, whenever the method is employed, the number of variables should not be large.

1.3 Artificial Neural Network

Since the artificial neural network (ANN) is going to be used to investigate the proposed algorithm, thus, the principle of ANN is introduced briefly. Generally, the ANN is categorized into feedforward and feedback networks[17]. In a feedforward network, information is propagated through the network in one direction until it emerges as the network's output. However, in a feedback (recurrent) network, the input information is propagated through the network but can also cycle back into the network (the signal is recurrent).

In this paper, the feedforward network is employed. Thus, the introduction of ANN will be focused on this type of network.

1.3.1 Multilayer Neural Network Architecture

In a typical three-layer feedforward neural network the first layer contains the input variables and is called the input layer. The last layer contains the output variables and is called the output layer. Layers in-between the input and output layers are called hidden layers; there can be more than one hidden layer. The processing unit elements are called nodes (Fig. 1-7): each of them is connected to the nodes of neighboring layers. The parameters associated with each of these connections are called weights [18].

The node (Fig. 1-8) sums the product of each connection weight (w_{jk}) from a node j to a node k and an input (x_j) to get the value SUM (see eq.1) for node k . This sum is simply the dot product of the input and weight vectors.

$$\text{sum}_k = \sum_j x_j w_{jk} + \gamma \quad (1)$$

It can be conveniently represented by matrix notation as

$$\text{sum} = [x^M][w^M] + [\gamma]^M \quad (2)$$

where M is the layer.

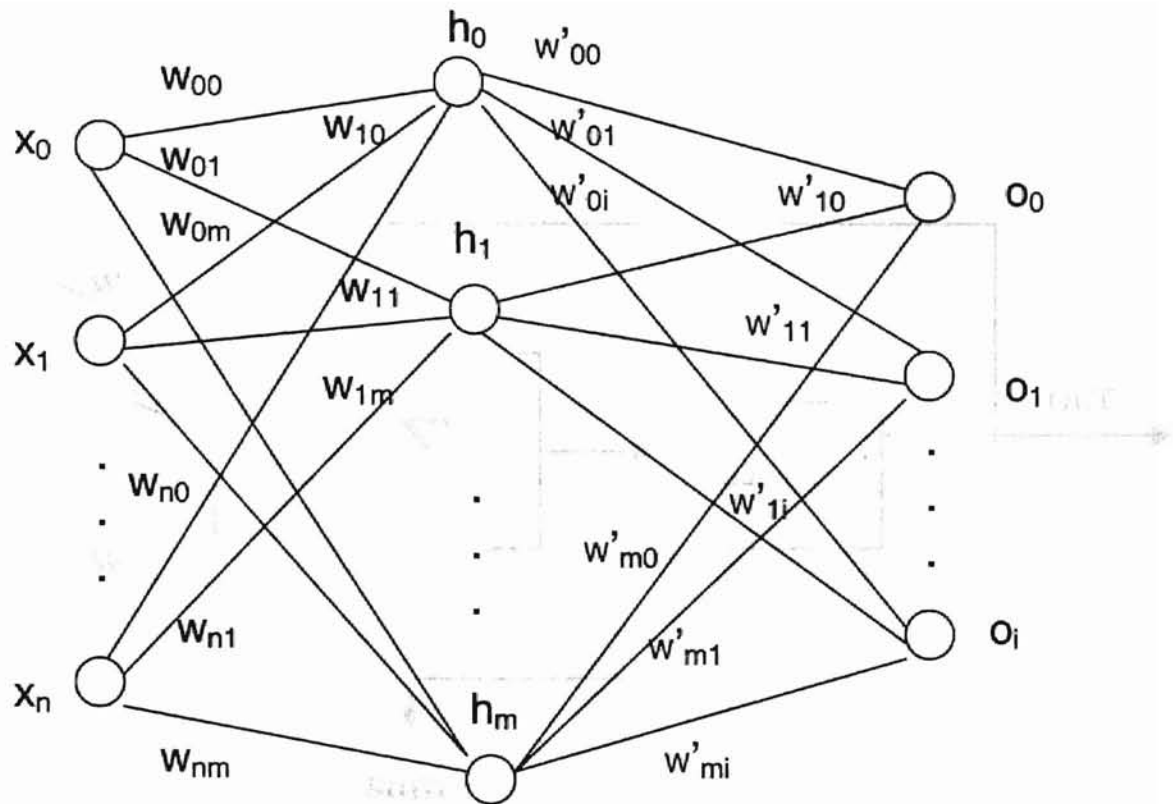


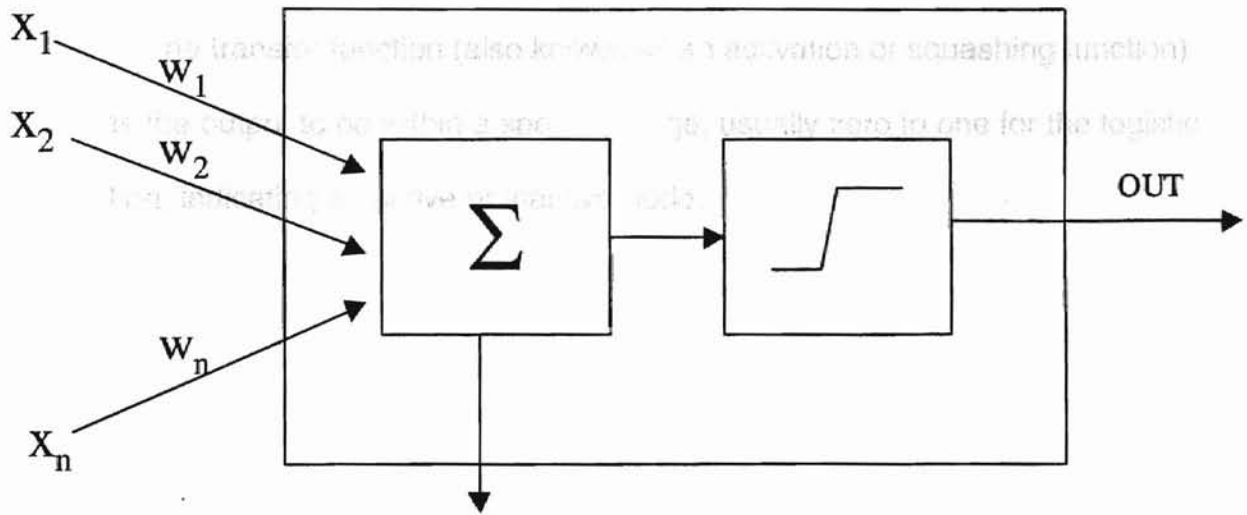
Figure 1-7. Three-layer neural network architecture. The first layer is input layer, h: hidden layer, the last layer is the output layer.

In vector notation, an additional dot product is used to give γ , which is called the bias value. The output of a bias j is always 1.0, and the weight γ 's are treated in the same fashion as the w_{jk} 's. This additional set of weights gives the network added degrees of flexibility, which enables it to solve more difficult problems. The value SUM is then supplied to a transfer function and outputs a value OUT[19].

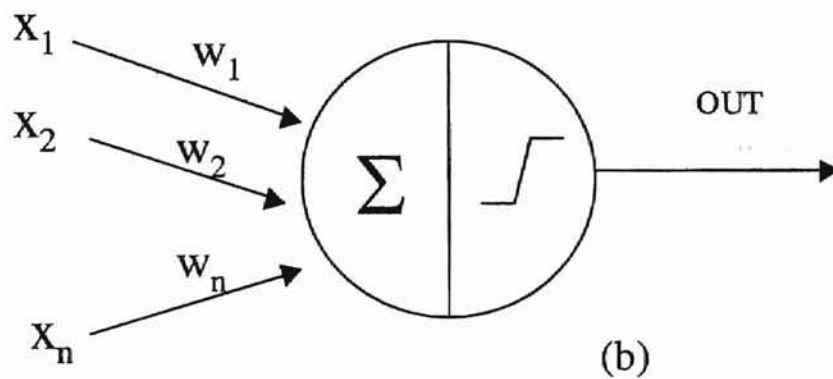
3.2.3 Transfer Function [20]

Any continuous and differentiable function may be used as a transfer function. However, the following logical function is most prevalent (S-shaped)

(a) (b) (c)



sum (a)



(b)

Figure 1-8. (a) is the composition of the neuron that will be represented by (b).

Any continuous and differentiable function may be used as a transfer function. However, the following logistical function is most prevalent (S-shaped).

$$f(x) = 1/(1+e^{-x})$$

The transfer function (also known as an activation or squashing function) forces the output to be within a specific range, usually zero to one for the logistic function, indicating an active or inactive node.

METHODOLOGY

2.1 Research Background

The genetic algorithm is a robust technique, based on the natural selection and genetic production mechanism. It processes a group or population of possible solutions within a search space. The search is probability guided, rather than deterministic or random searching, which distinguish it from traditional methods. The performance of the previous population guides the next generation. A complex search often involves a trade-off between exploiting the best solutions and robustly exploiting the space. The genetic algorithm is such a robust technique. However, robustness is not enough. When the vicinity of the global optimum has been located. A more powerful local tuning method is expected. Nelder and Mead presented a local searching technique, which is widely accepted. Compared to the genetic algorithm, this technique is well tuned for local searching but is not as robust as a genetic algorithm. Thus the idea is that hybridization of these two methods may improve the efficiency of the optimization.

2.2 Combination with the Simplex Downhill Method

It is widely believed that genetic algorithms are good at global optimization, but bad at fine, detailed local searching [21-24]. Some researchers suggest optimizing in two steps: first, using genetic algorithms to locate the area where the global optimum is, and then using other methods for further tuning. A

combination with the Nelder and Mead downhill method is the strategy we are going to explore. These two operations do not change the shape. The simplex downhill method presented by Nelder and Mead involves three basic operations: expansion, reflection, and contraction. Expansion is enlarging the particular search area. By reflection, a new point located on the other side of the worst point across the centroid of the remaining points is generated. Contraction is to select the point halfway between the worst point and the centroid. By repeatedly generating new points using one of the three basic operations, this simple method finds its way downhill to converge toward an optimum. However, Virginia J. Torczon [16] pointed that the Nelder-Mead simplex algorithm only rescales the entire simplex as a last resort. If no improvement can be found by taking any other step, the algorithm takes a “shrink” step, which is equivalent to the contraction step of the multi-dimensional search algorithm [16]. According to her claim, if the original simplex and its reflection are in a region where the function is convex, then the Nelder-Mead simplex algorithm will not consider the shrink step. Thus, in this search, the contraction step is avoided.

The genetic algorithm can help us to locate the most promising area. The use of the downhill method is intended to speed up the search when a promising area has been found when the number of parameters is less than 8 to 16 according to the investigation of Virginia J. Torczon. However, the restriction is released since only reflection and averaging are used in the simplex method. When selecting the next population of points based on the previous population of

points, we first consider the reflection operation. If the reflection does not work, however, we choose averaging. These two operations don not change the shape of the simplex, and thus avoid the problem pointed out by Torczon.

The simplex downhill method is not globally optimal but has a better local tuning property. If the genetic algorithm can be combined with it, improvement in performance can be expected. The combination can be made by generating part of the new points by the simplified genetic algorithm and part of them by the downhill method. The proportion of points generated by the two methods can be constant or is allowed to vary as the global optimum is approached. From the beginning of the search, a very low proportion of points are from the global optimum. As the search progress, the points will gradually approach the global optimum and then a high proportion of points generated by the downhill method is needed to speed up convergence.

2.3 Consideration of Genetic Diversity

Genetic diversity is very important for genetic algorithms. The loss of diversity means premature convergence and failure to achieve the global optimum. Community size and mutation probability can increase diversity and lead to global optimizing at the expense of slowing the procedure and taking more time. The proposed above guidelines, such as the one-couple, one-child policy, can avoid to some extent the loss of genetic diversity. A more efficient procedure is introduced by considering the distance among the points to purge the unwanted candidates and maintain a certain degree of diversity [25].

To measure diversity, the Euclidean distance between two points

$$d = \left[\sum_{i=1}^n (x_i - y_i)^2 \right]^{1/2}$$

is used. Where x_i and y_i are the i th values of points x and y , respectively. It is evident that the bigger the value of d , the further the distance between the two points. For instance, $d = 0$ means that the two points are identical, that is, there is no difference between them. Obviously, to keep one of them in the community is enough. When d is very close to zero, the two points are almost identical; if they produce a new point, this new point must be very close to their parents and is unlikely to bring much further improvement, unless they are close to the global optimum. Therefore, the distance d from the best-so-far point can be considered as a factor to save some of the promising candidates and improve the performance of the algorithm.

2.4 Basics of Simple Genetic Algorithm with Simplex Local Tuning

Objective function:

$$f = f(x_1, x_2, \dots, x_n)$$

$$a_i \leq x_i \leq b_i, i = 1, 2, 3, \dots, n$$

where x_i is the i th variable and a_i and b_i are the limits of that variable. The algorithm is summarized as follows:

Step 1) Initialization: randomly select m distinct points of that variable from the search space with equal probability. That is, generate a random number subject to a uniform (0, 1) distribution. Calculate the value of the first variable by

$$x_i = a_i + r_i(b_i - a_i)$$

Do the same for the other variables. These steps identify the first point in the search space. Repeat this procedure m times.

Step 2) Fitness Evaluation: Calculate the objective function values of the m points. P_1, P_2, \dots, P_m (2.2)

Step 3) Point ranking: Sort the m points in order of descending objective function value, so that the first point represents the worst and the last point represents the best. k is the maximum number of generations expected. The rest of

Step 4) Assigning probabilities to each point: Each of the points is assigned a probability p_i , $i = 1, 2, \dots, m$, giving higher probabilities to the points with lower function value and lower probabilities to those with higher function values. Thus, the best point has the highest probability p_1 , while the worst point has the lowest probability p_m . The other points have probabilities ranging from p_m to p_1 . The following linear relation can be used.

$$p_i = p_m + [(i - 1)/(m - 1)](p_1 - p_m), \quad i = 1, 2, 3, \dots, m - 1$$

Probability is nonnegative and the total probability should sum to one.

Step 5) Selecting parents: randomly select two points from the m points according to the probability p_i . Make sure that the two points are not identical.

Step 6) Crossover: For each of the genes (variables), randomly select one value from the corresponding two selected points to construct a new point.

Step 7) Mutation: Occasionally, with a small probability p_m , alter the newly created point. To do this, for each of the genes of the newly created point, generate a random number r , if $p_m > r$, replace the value of that gene by another uniformly distributed (0, 1) random number.

Step 8) Repeat k times Step 5 ~ 7 so that k new points are generated; k is a number between 0 and m , which is controlled by the following equations:

Step 1) $P_d = k/k_m$ family construct a new point (2-1) in the feasible space

Step 1) $P_g = 1 - P_d$ family construct a new point (2-2) the size of the new generation is

where P_g and P_d are the proportion of the points generated by the genetic algorithm and by the downhill method, respectively. k is the generation sequential number, and k_m is the maximum number of generations expected. The rest of the offspring will be generated by a simplex downhill method. (Step 1 ~ Step 8 are the genetic method)

Step 9) Construct a subcommunity: Randomly select s points from the m old points according to their probabilities to construct a subcommunity.

Step 10) Check whether this subcommunity is identical to any one of the previous constructed subcommunities of this generation. If it is, go back to Step 9 again; otherwise, go to Step 11.

Step 11) Compute the centroid of them without including the worst point, that is,

$$X_{ic} = [1/(s - 1)] \sum_{j=1}^{s-1} x_{ij}, \quad i = 1, 2, 3, \dots, n.$$

Step 12) Construct a new point by reflecting the worst point through the centroid point, that is,

$$y_{ij} = 2x_{ic} - x_{is}, \quad i = 1, 2, 3, \dots, n.$$

where x_{is} is the worst one of the s points. Then, evaluate the objective function value. If this point is better than the worst point of the older generation, then go to Step 15; otherwise, go to Step 13.

Step 13) Use the centroid point as the new point and evaluate its objective function value. If this point is still not better than the worst point, go to Step 14; otherwise go to Step 15.

Step 14) Randomly construct a new point within the feasible space.

Step 15) Repeat $m - k$ times Step 9 ~ 14 until the size of the new generation is the same size as that of their parents m . (Step 9 ~ Step 15 are downhill simplex method)

Step 16) These new points produced by the two methods represent the offspring population and are going to compete with their parents.

Step 17) Sort the newly created points into descending order.

Step 18) If the best point of the new generation (the last one) is not better than the best one of the old generation, then replace the worst point of the new generation by the best point of the old generation and re-sort them. This step is to ensure that the current best-so-far point in the community is always retained.

Step 19) Start from the second-best point of the new generation and compare it with the point in the same rank of the old generation. If the new point is better than the old one and is farther away from the best-so-far point, then keep the new one and discard the old one; then compare the rest until they are all finished. Go to Step 22; otherwise, go to Step 20.

Step 20) If the distance of the old point is farther away from the best-so-far point than the new one and has better fitness, then keep the old one and reject the new one and go to Step 19 to screen others; otherwise, go to Step 21.

Step 21) If the distance of the new one from the best-so-far point d_n times the objective function value of the old one f_0 is greater than the distance of the old one d_0 times the objective function value of the new one f_n , (i.e., $d_n f_0 > d_0 f_n$), then select the new one and go to Step 19, otherwise, generate a random number. If

it is greater than 0.5, then keep the old one and discard the new one and vice versa.

Step 22) Repeat Step 5 ~ 21 until either a predetermined iterative number or an acceptable objective function value is reached.

The neural network structure is employed for investigating the above algorithm and the program is written in C++ language. The objective function has been restricted to:

$$S = \left(\sum_{i=1}^n (y_i - x_i)^2 \right)^{1/2}$$

where y is computed output, x is experimental output. The purpose of training the neural network is to minimize the objective function by adjusting the weights during the program iterations with the proposed algorithm. This strategy of minimizing the objective function value is maintained for all cases in this paper.

RESULTS AND DISCUSSION

3.1 Case 1. Minimize a simple, single-minimum function,

$$f(x_1, x_2, x_3, \dots, x_{10}) = \sum_{i=1}^{10} x_i^2,$$

$$-2 < x_i < 10, i = 1, 2, 3, \dots, 10.$$

This function has a unique minimum at $x_i = 0$. This problem can be solved by many methods efficiently. In order to demonstrate the improvement provided by the proposed genetic algorithm, a comparison between using and not using genetic diversity measurements was conducted.

When the genetic diversity measurement is not used, the genetic algorithm simply produces one generation after another, following the traditional genetic algorithm method. The proposed simplified genetic algorithm method is as follows. With the mutation probability $p_m = 0.10$, different community sizes are used and the ten-run-average best-so-far objective function value is calculated for various numbers of objective function evaluations.

Table 1 shows two attractive advantages of using diversity guidance. Firstly, performance is different when genetic diversity guidance is used: the efficiency of the genetic algorithm is remarkably improved. 200 epochs with genetic diversity guidance produces a much better result than 3200 epochs without genetic diversity guidance.

Table 1: Comparison of the impact of using genetic diversity guidance for the (SD) in a simplified genetic algorithm, case 1, $p_m = 0.1$.

m	200 epochs		3200 epochs	
	(a)	(b)	(a)	(b)
5	123.48	2.16	117.89	0.024
10	63.09	2.00	57.94	0.020
20	38.34	1.77	34.92	0.034
90	17.44	3.70	13.98	0.69

m : Community size; (a): Not using genetic diversity guidance; (b): Using genetic diversity guidance.

Secondly, when genetic diversity guidance is introduced, the genetic algorithm prefers a smaller community size, rather than a bigger one. When the community size is large enough, the efficiency of the genetic diversity guidance is damped because a large community size can contain almost every possible character. That may be the reason why traditional genetic algorithms need a very large community size. However, as the search progresses, all points converge gradually to the global minimum. Not considering diversity guidance can result in many identical or semi-identical points in the community and can slow down the approach to the global minimum. Therefore, no matter how large the community size is, the introduction of diversity guidance can improve the efficiency of the genetic algorithm.

3.2 Case 2. Genetic algorithm (GA) or Combination with the Downhill Method (GD) in Application of Ion-selective Electrodes of Analytical Chemistry.

For the data listed in the Table 2, set $p_m = 0.10$ and use Eqs (2-1) and (2-2) to control the portion of new points generated by the genetic algorithm and downhill method. The objective function $S = (\sum_{i=1}^n (y_i - x_i)^2)^{1/2}$ (where y is computed output, x is experimental output) is minimized for each iteration.

Table 2. Training set for the $K^+/Ca^{2+}/NO_3^-/Cl^-$ system [26].

No.	Conc. ($mmol^{-1}$)				Potential (mv)			
	K	Ca	NO_3	Cl	K	Ca	NO_3	Cl
1	0.100	0.100	0.100	0.300	-147.8	-22.5	228.2	227.0
2	0.990	0.099	0.099	1.188	-90.8	-23.3	227.2	195.5
3	9.991	0.090	0.090	10.171	-31.4	-27.4	209.8	141.2
4	0.498	0.498	0.100	1.493	-109.8	-3.3	226.5	189.1
5	4.988	0.475	0.095	5.938	-49.6	-5.6	217.12	154.5
6	0.100	0.999	0.100	2.098	-146.9	3.0	219.4	180.5
7	0.990	0.990	0.099	2.970	-91.6	4.3	219.3	172.2
8	9.991	0.900	0.90	11.791	-31.4	0.8	204.6	137.7
9	0.498	5.224	0.100	10.945	-110.8	23.2	205.6	140.4
10	4.988	4.988	0.095	14.964	-50.8	22.0	201.2	132.2
11	0.100	11.089	0.100	22.278	-150.3	27.6	194.8	123.2

12	0.990	10.990	0.099	22.970	-93.8	30.6	196.5	122.5
13	9.991	9.991	0.090	29.973	-33.6	28.9	190.8	115.1
14	0.498	0.100	0.498	0.697	-109.3	-25.2	188.4	207.5
15	4.988	0.095	0.475	5.178	-50.8	-27.4	187.9	157.6
16	0.100	0.500	0.500	1.099	-146.1	-5.8	187.6	196.5
17	0.990	0.495	0.495	1.980	-92.3	-5.1	188.6	181.2
18	9.991	0.450	0.450	10.891	-32.3	-8.5	185.9	138.5
19	0.498	0.995	0.498	2.488	-109.0	3.5	188.6	176.6
20	4.988	0.950	0.475	6.888	-50.5	2.1	187.6	150.8
21	0.100	5.245	0.500	10.589	-147.6	20.7	184.2	143.4
22	0.990	5.198	0.495	11.386	-93.1	21.5	185.2	140.9
23	9.991	4.725	0.450	19.442	-32.6	18.8	182.5	125.2
24	0.498	11.045	0.498	22.587	-111.0	29.1	180.0	122.5
25	4.988	10.546	0.475	26.081	-51.5	29.3	179.5	118.3
26	0.100	0.100	0.999	0.300	-142.5	-24.2	169.9	227.9
27	0.990	0.099	0.990	1.188	-91.1	-24.0	170.9	195.5
28	9.991	0.090	0.900	10.171	-29.9	-27.7	171.4	140.0
29	0.498	0.498	0.995	1.493	-108.3	-4.1	171.7	185.8
30	4.988	0.475	0.950	5.938	-48.1	-5.8	171.9	152.5
31	0.100	5.245	0.999	10.589	-145.2	22.0	170.9	140.9
32	0.990	5.198	0.990	11.386	-92.8	22.5	171.4	138.7
33	9.991	4.725	0.900	19.442	-32.6	20.3	171.2	124.7
34	0.498	0.995	0.995	2.488	-107.6	3.8	171.4	175.1

35	4.988	0.950	0.950	6.888	-48.1	2.6	172.2	156.7
36	0.100	11.089	0.999	22.278	-144.9	30.8	169.2	122.0
37	0.990	10.990	0.990	22.970	-92.3	30.8	169.4	121.3
38	9.991	9.991	0.900	29.973	-32.6	29.1	168.7	113.9
39	0.498	0.100	5.224	0.697	-105.6	-26.9	129.9	207.5
40	4.988	0.095	4.988	5.178	-50.0	-27.9	132.2	157.9

At the very beginning, we would like to show the over_fit which happens when the number of data training set points is less than the number of parameters of the neural network. Thus the K^+ concentrations and their corresponding potentials in Table 2 were chosen to train the network. The neural network architecture is three layers with one neuron in the input layer, three neurons in the hidden layer, and one neuron in the output layer. It seems that we have 40 data in the training data, however, the concentrations of K^+ only have five distinguished data set points, re-measured eight times each. In contrast, there are six parameters in the neural network. The neural network generation was trained using only the K^+ data set. Figure 3-1 depicts the over-fit phenomenon. The range of input potential is from -160 mv to -20 mv (Table 3). When the output is plotted against input data, the over-fit is observed apparently specially when the potential is quite low. For instance, when the potential is around -150 mv, the slope is extremely large. Then turning up of the fitted curve at low values of the potential is not supported by the data.

Table 3. Generation data for over-fit

Potential (mv)	Log(K)	Potential (mv)	Log(K)	Potential (mv)	Log(K)	Potential (mv)	Log(K)
-160	0.1995	-124	-0.5926	-88	0.0882	-52	0.6829
-159	-0.0628	-123	-0.5726	-87	0.1060	-51	0.6981
-158	-0.3055	-122	-0.5527	-86	0.1236	-50	0.7133
-157	-0.5012	-121	-0.5328	-85	0.1412	-49	0.7284
-156	-0.6519	-120	-0.5130	-84	0.1587	-48	0.7435
-155	-0.7654	-119	-0.4932	-83	0.1761	-47	0.7585
-154	-0.8493	-118	-0.4735	-82	0.1935	-46	0.7734
-153	-0.9101	-117	-0.4538	-81	0.2108	-45	0.7882
-152	-0.9527	-116	-0.4342	-80	0.2280	-44	0.8030
-151	-0.9811	-115	-0.4147	-79	0.2452	-43	0.8178
-150	-0.9984	-114	-0.3952	-78	0.2623	-42	0.8324
-149	-1.0062	-113	-0.3758	-77	0.2793	-41	0.8470
-148	-1.0093	-112	-0.3565	-76	0.2963	-40	0.8615
-147	-1.0062	-111	-0.3372	-75	0.3131	-39	0.8760
-146	-0.9990	-110	-0.3180	-74	0.3300	-38	0.8903
-145	-0.9888	-109	-0.2988	-73	0.3467	-37	0.9047
-144	-0.9761	-108	-0.2797	-72	0.3634	-36	0.9189
-143	-0.9616	-107	-0.2607	-71	0.3800	-35	0.9331
-142	-0.9456	-106	-0.2418	-70	0.3966	-34	0.9472
-141	-0.9286	-105	-0.2229	-69	0.4131	-33	0.9613

-140	-0.9107	-104	-0.2040	-68	0.4295	-32	0.9753
-139	-0.8921	-103	-0.1853	-67	0.4458	-31	0.9892
-138	-0.8731	-102	-0.1666	-66	0.4621	-30	1.0030
-137	-0.8537	-101	-0.1479	-65	0.4783	-29	1.0168
-136	-0.8340	-100	-0.1294	-64	0.4945	-28	1.0305
-135	-0.8142	-99	-0.1109	-63	0.5105	-27	1.0442
-134	-0.7941	-98	-0.0924	-62	0.5265	-26	1.0578
-133	-0.7740	-97	-0.0741	-61	0.5425	-25	1.0713
-132	-0.7539	-96	-0.0778	-60	0.5583	-24	1.0848
-131	-0.7337	-95	-0.0375	-59	0.5741	-23	1.0962
-130	-0.7135	-94	-0.0193	-58	0.5900	-22	1.1115
-129	-0.6933	-93	-0.0012	-57	0.6055	-21	1.1248
-128	-0.6731	-92	0.0168	-56	0.6211	-20	1.1380
-127	-0.6529	-91	0.0348	-55	0.6367		
-126	-0.6328	-90	0.0527	-54	0.6521		
-125	-0.6127	-89	0.0705	-53	0.6675		

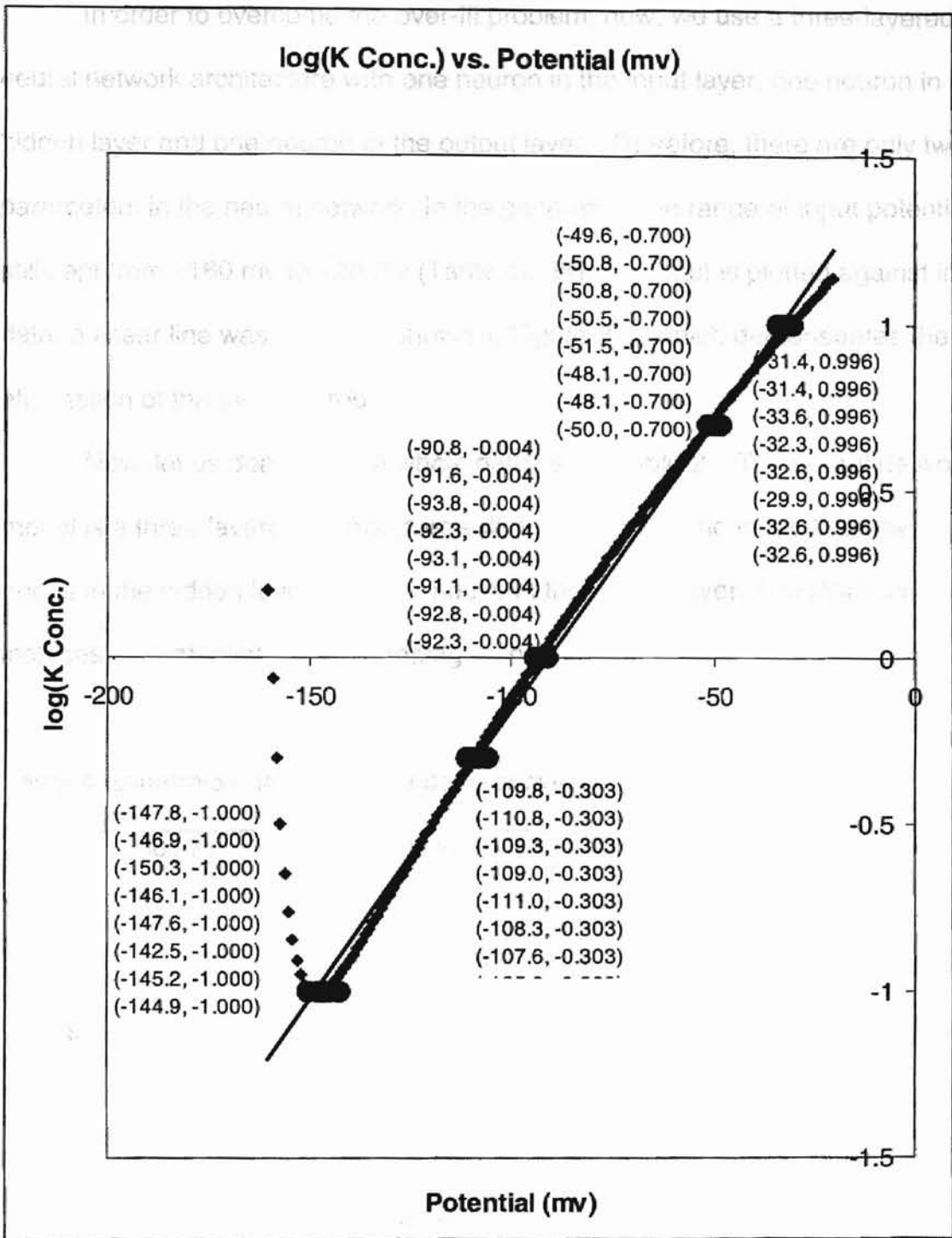


Figure 3-1. Over-fit phenomenon.

-152 In order to overcome the over-fit problem, now, we use a three-layered 054
 neural network architecture with one neuron in the input layer, one neuron in the
 hidden layer and one neuron in the output layer. Therefore, there are only two 00
 parameters in the neural network. In the generation the range of input potential is
 still kept from -160 mv to -20 mv (Table 4). When output is plotted against input
 data, a linear line was obtained shown in Figure 3-2, which demonstrates the 017
 elimination of the over-fit problem. 74 0.2020 78 0.3082

-138 Now, let us deal with the whole data set in Table 2. The neural network 02
 model is a three-layered architecture with four nodes in the input layer, two 0405
 nodes in the hidden layer and four nodes in the output layer. The input data set
 includes the potentials corresponding to their concentrations.

Table 4 Generation data for elimination of over-fit

Potential (mv)	log(K)	Potential (mv)	log(K)	Potential (mv)	log(K)	Potential (mv)	log(K)
-160	-1.1961	-124	-0.5749	-88	0.0462	-52	0.6674
-159	-1.1789	-123	-0.5577	-87	0.0635	-51	0.6846
-158	-1.1616	-122	-0.5404	-86	0.0807	-50	0.7019
-157	-1.1443	-121	-0.5232	-85	0.0980	-49	0.7191
-156	-1.1271	-120	-0.5059	-84	0.1152	-48	0.7364
-155	-1.1098	-119	-0.4887	-83	0.1325	-47	0.7537
-154	-1.0926	-118	-0.4714	-82	0.1497	-46	0.7709
-153	-1.0753	-117	-0.4542	-81	0.1670	-45	0.7882

-152	-1.0581	-116	-0.4369	-80	0.1843	-44	0.8054
-151	-1.0408	-115	-0.4197	-79	0.2015	-43	0.8227
-150	-1.0236	-114	-0.4024	-78	0.2188	-42	0.8400
-149	-1.0063	-113	-0.3851	-77	0.2360	-41	0.8572
-148	-0.9891	-112	-0.3679	-76	0.2533	-40	0.8744
-147	-0.9718	-111	-0.3506	-75	0.2705	-39	0.8917
-146	-0.9545	-110	-0.3334	-74	0.2878	-38	0.9089
-145	-0.9373	-109	-0.3161	-73	0.3050	-37	0.9262
-144	-0.9200	-108	-0.2989	-72	0.3223	-36	0.9435
-143	-0.9028	-107	-0.2816	-71	0.3395	-35	0.9607
-142	-0.8855	-106	-0.2644	-70	0.3568	-34	0.9780
-141	-0.8683	-105	-0.2471	-69	0.3741	-33	0.9952
-140	-0.8510	-104	-0.2299	-68	0.3913	-32	1.0125
-139	-0.8338	-103	-0.2126	-67	0.4086	-31	1.0297
-138	-0.8165	-102	-0.1954	-66	0.4258	-30	1.0470
-137	-0.7993	-101	-0.1781	-65	0.4431	-29	1.0642
-136	-0.7820	-100	-0.1608	-64	0.4603	-28	1.0815
-135	-0.7647	-99	-0.1439	-63	0.4776	-27	1.0987
-134	-0.7475	-98	-0.1263	-62	0.4948	-26	1.1160
-133	-0.7302	-97	-0.1091	-61	0.5121	-25	1.1333
-132	-0.7130	-96	-0.0918	-60	0.5293	-24	1.1505
-131	-0.6957	-95	-0.0746	-59	0.5466	-23	1.1678
-130	-0.6785	-94	-0.0573	-58	0.5639	-22	1.1850

-129	-0.6612	-93	-0.0401	-57	0.5811	-21	1.2023
-128	-0.6440	-92	-0.0228	-56	0.5984	-20	1.2195
-127	-0.6267	-91	-0.0056	-55	0.6156	-19	1.2367
-126	-0.6095	-90	0.0117	-54	0.6329	-18	1.2539
-125	-0.5922	-89	0.0290	-53	0.6501	-17	1.2711

Table 5 shows clearly that a combination with the downhill method can further improve the efficiency of the genetic algorithm, especially when a more accurate result is expected. This is because the genetic algorithm only drives the points into the vicinity of the global minimum. The rest of the work is left for the downhill method to finish.

Table 6 shows the data obtained from the network using the proposed algorithm compared to the experimental data.

With the weights from the above training algorithms, for test data in Table 7 the test results are presented in Table 8.

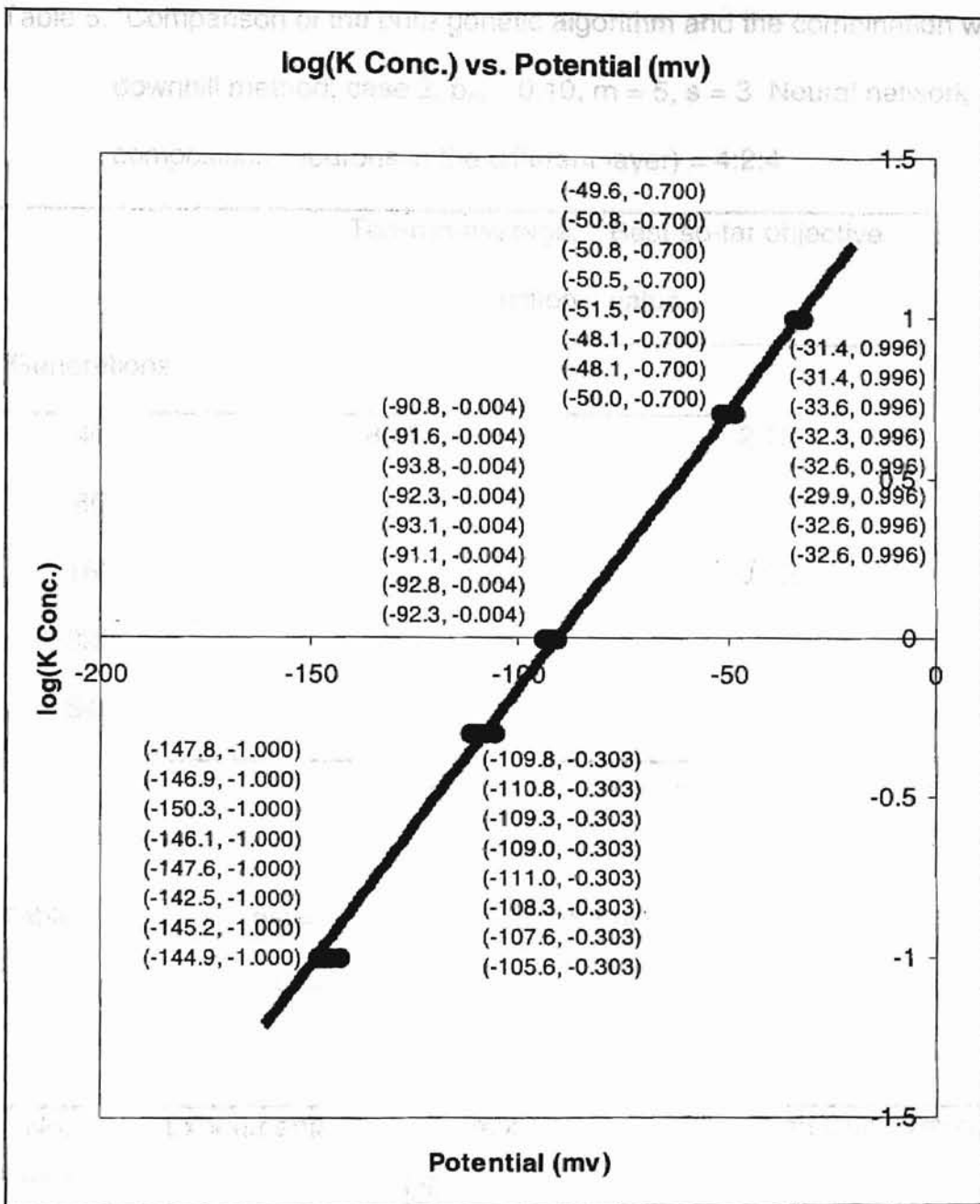


Figure 3-2. Elimination of over-fit.

Table 5. Comparison of the pure genetic algorithm and the combination with downhill method, case 2, $p_m = 0.10$, $m = 5$, $s = 3$. Neural network composition (neurons in the different layer) = 4:2:4.

Generations	GA	GD
40	8.19	2.15
80	3.84	1.56
160	2.47	0.95
320	1.76	0.53
640	1.03	0.28

Table 6. Training result of data set in Table 2 using GD method, case2, $p_m = 0.2$, $m = 5$, $s = 3$, epochs = 1000000, neural network composition = 4:2:4. Objective function value is 0.061.

No.	Experimental Conc. (mmol^{-1})				Computed Conc. (mmol^{-1})			
	K	Ca	NO_3	Cl	K	Ca	NO_3	Cl
1	0.100	0.100	0.100	0.300	0.100	0.100	0.103	0.303
2	0.990	0.099	0.099	1.188	0.940	0.099	0.101	1.188
3	9.991	0.090	0.090	10.171	9.881	0.097	0.105	9.907
4	0.498	0.498	0.100	1.493	0.524	0.515	0.078	1.466
5	4.988	0.475	0.095	5.938	5.172	0.463	0.096	5.769

6	0.100	0.999	0.100	2.098	0.100	1.037	0.079	2.113
7	0.990	0.990	0.099	2.970	0.948	1.008	0.094	2.714
8	9.991	0.900	0.90	11.791	9.397	0.900	0.916	12.615
9	0.498	5.224	0.100	10.945	0.517	5.816	0.088	11.660
10	4.988	4.988	0.095	14.964	5.144	5.145	0.105	15.07
11	0.100	11.089	0.100	22.278	0.100	11.063	0.104	21.773
12	0.990	10.990	0.099	22.970	1.035	10.393	0.095	22.716
13	9.991	9.991	0.090	29.973	9.255	10.012	0.089	29.617
14	0.498	0.100	0.498	0.697	0.487	0.094	0.504	0.702
15	4.988	0.095	0.475	5.178	3.306	0.095	0.489	4.948
16	0.100	0.500	0.500	1.099	0.100	0.500	0.535	1.100
17	0.990	0.495	0.495	1.980	0.922	0.479	0.599	1.932
18	9.991	0.450	0.450	10.891	9.375	0.408	0.456	10.765
19	0.498	0.995	0.498	2.488	0.474	1.015	0.474	2.341
20	4.988	0.950	0.475	6.888	4.761	0.944	0.445	7.150
21	0.100	5.245	0.500	10.589	0.100	4.919	0.497	10.512
22	0.990	5.198	0.495	11.386	0.986	4.463	0.491	11.635
23	9.991	4.725	0.450	19.442	9.204	4.447	0.444	20.012
24	0.498	11.045	0.498	22.587	0.498	11.043	0.489	21.670
25	4.988	10.546	0.475	26.081	5.445	10.038	0.468	26.105
26	0.100	0.100	0.999	0.300	0.100	0.096	1.090	0.269
27	0.990	0.099	0.990	1.188	0.922	0.100	0.992	1.184
28	9.991	0.090	0.900	10.171	9.978	0.099	0.934	10.735

29	0.498	0.498	0.995	1.493	0.459	0.541	0.979	1.665
30	4.988	0.475	0.950	5.938	5.354	0.499	0.895	6.051
31	0.100	5.245	0.999	10.589	0.100	5.280	1.050	11.040
32	0.990	5.198	0.990	11.386	0.990	5.231	0.931	12.004
33	9.991	4.725	0.900	19.442	9.141	5.189	0.849	20.153
34	0.498	0.995	0.995	2.488	0.509	1.057	0.997	2.493
35	4.988	0.950	0.950	6.888	4.828	0.939	0.964	6.824
36	0.100	11.089	0.999	22.278	0.100	11.120	1.083	21.866
37	0.990	10.990	0.990	22.970	1.054	10.453	1.005	22.608
38	9.991	9.991	0.900	29.973	9.357	10.105	0.903	29.903
39	0.498	0.100	5.224	0.697	0.525	0.092	5.187	0.724
40	4.988	0.095	4.988	5.178	5.150	0.097	4.990	4.996

Table 7. Test data set for the $K^+/Ca^{2+}/NO_3^-/Cl^-$ system [26].

No.	Experimental Conc. ($mmol^{-1}$)				Potential (mv)			
	K	Ca	NO_3	Cl	K	Ca	NO_3	Cl
1	0.100	0.100	0.100	0.300	-147.8	-22.5	228.2	227.0
2	0.990	0.099	0.099	1.188	-90.8	-23.3	227.2	195.5
3	9.991	0.090	0.090	10.171	-31.4	-27.4	209.8	141.2
4	0.498	0.498	0.100	1.493	-109.8	-3.3	226.5	189.1
5	4.988	0.475	0.095	5.938	-49.6	-5.6	217.12	154.5
6	0.100	0.999	0.100	2.098	-146.9	3.0	219.4	180.5
7	0.990	0.990	0.099	2.970	-91.6	4.3	219.3	172.2

8	9.991	0.900	0.90	11.791	-31.4	0.8	204.6	137.7
9	0.498	5.224	0.100	10.945	-110.8	23.2	205.6	140.4
10	4.988	4.988	0.095	14.964	-50.8	22.0	201.2	132.2

Table 8. Test result for data set in Table 7 using GD method, case2, $p_m = 0.2$, $m = 5$, $s = 3$, neural network composition = 4:2:4. Objective function value is 0.056.

No.	Experimental Conc. (mmol^{-1})				Computed Conc. (mmol^{-1})			
	K	Ca	NO_3	Cl	K	Ca	NO_3	Cl
1	0.100	0.100	0.100	0.300	0.100	0.100	0.100	0.303
2	0.990	0.099	0.099	1.188	0.940	0.099	0.098	1.185
3	9.991	0.090	0.090	10.171	9.881	0.097	0.090	9.908
4	0.498	0.498	0.100	1.493	0.524	0.515	0.100	1.466
5	4.988	0.475	0.095	5.938	5.172	0.463	0.096	5.769
6	0.100	0.999	0.100	2.098	0.100	1.037	0.100	2.113
7	0.990	0.990	0.099	2.970	0.948	1.008	0.098	2.714
8	9.991	0.900	0.900	11.791	9.797	0.890	0.920	11.620
9	0.498	5.224	0.100	10.945	0.516	5.816	0.100	10.866
10	4.988	4.988	0.095	14.964	5.143	5.145	0.093	15.074

In this case, we would like to examine sensitivity of parameters on the efficiency of the genetic with downhill method. There are a few parameters in the

proposed algorithm that need to be determined. Some of them are for encoding, while others are for the algorithm itself. The proposed algorithm has three parameters of its own, they are the community size m (the number of points for each generation), the mutation probability p_m , and the subcommunity size s . Case 1 shows some general guidelines for the selection of these three parameters. However, the selection of the parameters may be problem-related.

Parameters suitable for the problem in case 1 may not be suitable for the problem in case 2, but there must be some general guidelines. Some researchers have discussed the selection of parameters of traditional genetic algorithms, on the other hand, their results are unlikely to be suitable for the proposed algorithm because of improvements and modifications.

To investigate the sensitivity of the parameters for the proposed algorithm, various parameters are used and again the ten-run-average, best-so-far objective function values are calculated. The results of 800 and 1600 epochs of objective function evaluations (Table 9) show the following points:

- (a) The proposed algorithm for Case 2 has a performance almost similar to that for the former cases, and it is not so sensitive to the parameters. Therefore, it is a very robust algorithm and can be expected to be successfully used in many conditions.
- (b) The algorithm is not too sensitive to community size and subcommunity size, though it seems to prefer smaller values. The values $m = 5$ and $s = 3$ yields the best performance.

4.3. (c) 3. The algorithm is relatively more sensitive to mutation probability p_m

The than the other two parameters m and s . Its careful selection can

extended to improve the performance of the algorithm. The experimental

data in Table 10 [27] are the viscosity measurements in different temperatures

Table 9. Sensitivities to the parameters of the proposed algorithm, Case 2, ten-

days of run-average, best-so-far objective function values. is in the hidden layer

Sensitivity to mutation probability p_m for $m = 5, s = 3$							
p_m	0.00	0.01	0.05	0.10	0.20	0.30	0.50
800 epochs	7.26	1.48	1.04	0.82	0.99	1.06	1.18
1600 epochs	5.38	1.24	0.97	0.68	0.82	0.88	0.98
Sensitivity to community size m for $p_m = 0.10, s = 3$							
m	4	5	10	20			
800 epochs	1.34	1.35	1.04	1.14			
1600 epochs	0.93	0.90	0.99	0.97			
Sensitivity to subcommunity size s for $p_m = 0.10, m = 10$							
s	3	4	5	6			
800 epochs	1.24	1.32	1.27	1.71			
1600 epochs	0.99	1.27	1.21	1.03			

4.3. Case 3: Chemical Engineering. Viscosity at different temperatures and pressures

The investigation of the efficiency of the proposed algorithm is also extended to solve some problems in chemical engineering. The experimental data in Table 10 [27] are the viscosity measurements in different temperatures and pressures. In this investigation, the neural network architecture is a three-layered structure with two nodes in the input layer, five nodes in the hidden layer and one node in the output layer. The purpose of training the network is to minimize the objective function $S = \sum_{i=1}^n (y_i - x_i)^2$ (where y is computed output, x is experimental output) in each iteration.

For comparison, the genetic algorithm (notice that this is not a traditional GA, because of the modification) and the downhill search generating procedure (notice the difference with the Nelder and Mead method because the contraction and reflection are not used in the proposed algorithm) are also calculated. The above suggested parameters are used for the proposed algorithm. Furthermore, they also apply to the corresponding GA and downhill searching.

Table 11 shows that genetic algorithm with simplex downhill method (GD) can gradually reach the global minimum. As the generation grows, the probability of reaching the global minimum is increased. Using the modified GA alone enables the vicinity of the global minimum to be reached with increasingly probability as time goes on, while the downhill method cannot reach the global minimum at all. The downhill method is always trapped in one of the local minima. Tables 12 and 13 demonstrate the training, test and generation data using the proposed algorithm, respectively.

Table 10.3 Data for the lubricant viscosity at different temperatures and pressures

Temperature ($^{\circ}\text{C}$)	Pressure (atm)	$\ln[\text{viscosity}]$ (experimental)
0.0	1.0	5.106
0.0	740.8	6.387
0.0	1407.5	7.385
0.0	363.2	5.791
0.0	1.0	5.107
0.0	805.5	6.361
0.0	3907.5	11.927
0.0	4125.5	12.426
0.0	2572.0	9.156
25.0	1.0	4.542
25.0	805.0	5.825
25.0	1505.9	6.705
25.0	2340.0	7.716
25.0	422.9	5.298
25.0	5064.3	11.984
25.0	5280.9	12.444
25.0	3647.3	9.523
25.0	2813.9	8.345
37.8	516.8	5.173

37.8	1738.0	6.650
37.8	1008.7	5.807
37.8	2749.2	7.741
37.8	1375.8	6.232
37.8	191.1	4.661
37.8	1.0	4.298
37.8	4849.8	10.511
37.8	5605.8	11.822
37.8	6273.9	13.068
37.8	3636.7	8.804
37.8	1949.0	6.855
37.8	1298.5	6.119
98.9	1.0	3.381
98.9	686.0	4.458
98.9	1423.6	5.207
98.9	2791.4	6.291
98.9	4213.4	7.327
98.9	2103.7	5.770
98.9	402.2	4.088
98.9	1.0	3.374
98.9	2219.7	5.839
98.9	6344.2	8.914
98.9	7469.4	9.983

98.9	1407.5640.9	7.385	8.323.474
98.9	363.4107.9	5.791	7.132.732

Table 11. Performance of the proposed algorithm (GD), Case 3, neural network composition = 2:5:1

Ten-run-average, best-so-far objective function value				
generation	GA	Downhill	GD	
50	1.41633	3.99894	1.24233	
100	0.94665	3.88616	0.722244	
200	0.930796	3.73996	0.541633	
400	0.789369	3.50923	0.285179	
1000	0.700704	3.00807	0.158261	
2000	0.639371	2.89261	0.0724673	
4000	0.538636	2.88345	0.045308	

Table 12. Training data for the lubricant viscosity at different temperatures and pressures, $m = 5$, $s=3$, $p_m = 0.10$, epoch = 8000, neural network composition = 2:5:1, objective function value = 0.019, using the proposed algorithm.

Temperature ($^{\circ}\text{C}$)	Pressure (atm)	ln[viscosity] (experimental)	ln[viscosity] (calculated)
0.0	1.0	5.106	5.162
0.0	740.8	6.387	6.417

0.0	1407.5	7.385	7.474
0.0	363.2	5.791	5.732
0.0	1.0	5.107	5.162
0.0	805.5	6.361	6.532
0.0	3907.5	11.927	11.697
0.0	4125.5	12.426	11.997
0.0	2572.0	9.156	9.300
25.0	1.0	4.542	4.472
25.0	805.0	5.825	5.799
25.0	1505.9	6.705	6.754
25.0	2340.0	7.716	7.629
25.0	422.9	5.298	5.071
25.0	5064.3	11.984	12.010
25.0	5280.9	12.444	12.253
25.0	3647.3	9.523	9.619
25.0	2813.9	8.345	8.242
37.8	516.8	5.173	4.953
37.8	1738.0	6.650	6.655
37.8	1008.7	5.807	5.860
37.8	2749.2	7.741	7.583
37.8	1375.8	6.232	6.323
37.8	191.1	4.661	4.424
37.8	1.0	4.298	4.237

37.8	4849.8	10.511	10.958
37.8	5605.8	11.822	12.062
37.8	6273.9	13.068	12.621
37.8	3636.7	8.804	8.793
37.8	1949.0	6.855	6.830
37.8	1298.5	6.119	6.240
98.9	1.0	3.381	3.689
98.9	686.0	4.458	4.337
98.9	1423.6	5.207	5.856
98.9	2791.4	6.291	6.239
98.9	4213.4	7.327	6.944
98.9	2103.7	5.770	6.003
98.9	402.2	4.088	3.900
98.9	1.0	3.374	3.689
98.9	2219.7	5.839	6.046
98.9	6344.2	8.914	9.124
98.9	7469.4	9.983	9.957
98.9	5640.9	8.323	8.289
98.9	4107.9	7.132	6.873

Table 13. Test and generalization data for the proposed algorithm, Case 3,

neural network composition = 2:5:1.

3.708

7.729

Test data, objective function value = 0.0016			
Temperature ($^{\circ}\text{C}$)	Pressure (atm)	In[viscosity] (experimental)	In[viscosity] (calculated)
0.0	1407.5	7.385	7.408
0.0	3907.5	11.927	11.898
25.0	1505.9	6.705	6.605
25.0	5280.9	12.444	12.170
37.8	1375.8	6.232	6.160
37.8	3636.7	8.804	8.730
98.9	2791.4	6.291	6.508
98.9	7469.4	9.983	10.044
Generation data			
Temperature ($^{\circ}\text{C}$)	Pressure (atm)	In[viscosity] (experimental)	In[viscosity] (calculated)
0.0	1868.1	7.973	8.191
0.0	3285.1	10.473	10.968
25.0	1168.4	6.226	6.206
25.0	2237.3	7.574	7.458
25.0	4216.9	10.354	10.710
37.8	2922.9	7.967	7.772
37.8	4044.6	10.511	9.401

98.9	3534.8	CHAPTER IV	6.726	6.969
98.9	4937.7	CONCLUSIONS	7.768	7.729

It is well known that genetic algorithm is a method for global optimization.

The genetic algorithm has been used in many situations such as calibration of parameters in water quality models and hydrology. When the method is used to solve some highly nonlinear problems, such as the optimization of the parameters of water quality models, it usually leads to a satisfactory solution.

The genetic algorithm was used to calibrate the parameters of the water quality model. The genetic algorithm was used to calibrate the parameters of the water quality model. The genetic algorithm was used to calibrate the parameters of the water quality model. The genetic algorithm was used to calibrate the parameters of the water quality model.

The genetic algorithm was used to calibrate the parameters of the water quality model.

The genetic algorithm was used to calibrate the parameters of the water quality model.

The genetic algorithm was used to calibrate the parameters of the water quality model.

in

1998, 6, 1, 1, 1, 1

CHAPTER IV

CONCLUSIONS

It is well-known that genetic algorithm is a method for global optimizing. Therefore, this method has been used in many situations such as calibration of model parameters in water resource analysis and hydrology. When the method is used to solve some tough optimizing problems, such as the optimization of the parameters of artificial neural networks, it results in a satisfactory outcome.

From the traditional and classic genetic algorithm and simplex downhill method, Genetic algorithm with simplex downhill method (GD) was constructed with the combination of the two methods, inheriting the advantages of both algorithms. Experimentally, this algorithm was tested with different cases. Based on the experimental results, the following conclusions can be drawn.

- (1) The GD algorithm is reasonable, well grounded, correct, and effective.
- (2) The proposed algorithm is robust. It can avoid local minima, reach the global optimum efficiently, and quickly settle down.
- (3) The amount of computing time needed for the many iterations may be large and expensive and could be a financial constraint in some cases.

REFERENCES

1. Goldberg, D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning*; Addison-Wesley: Reading, MA, 1989.
2. Lucasius, C.B., de Weijer, A. P., Buydens, L. M. and Kateman, G., CFIT: A Genetic Algorithm for Survival of the Fittest, *Chemom. Intell. Lab. Syst.* 1993, 19, pp. 337-341.
3. Lucasius, C.B. and Kateman, G., On K-medoid Clustering of Large Data Sets with the Aid of Genetic Algorithm: Background, Feasibility and Comparison, *Chemom. Intell. Lab. Syst.* 1994, 25, pp. 99-145.
4. Hibbert, D.A., Genetic Algorithms in Chemistry, *Chemom. Intell. Lab. Syst.* 1993, 19, pp. 277-293.
5. Holland, J. H., Genetic Algorithms, *Scientific American*, July, 1992, pp.66-87.
6. Holland, J. H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.
7. Hecht-Nielsen R., *Neurocomputing*. Addison-Wesley, 1991.
8. Shaffer, R.E., *Optimization Methods for the Multivariate Analysis of Infrared Spectral and Interferogram Data*, Ph.D. Dissertation, Ohio University, Athens, OH, 1996.
9. Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.
10. Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag Berlin Heidelberg, 1996.

11. Davis, Lawrence. *Handbook of Genetic Algorithms*. Van Nostrand-Reinhold, 1991.
12. Box, G.E.P., Wilson, K.B., On the Experimental Attainment of Optimum Conditions, *J. Roy. Stat. Soc.* 1951, 13, pp. 1-45.
13. Box, G.E.P., Evolutionary Operation: A Method for Increasing Industrial Productivity, *Appl. Statist.* 1957, 6, pp. 81-101.
14. Spendley, W., Hext, G.R., Himsworth, F.R., Sequential Application of Simplex Designs in Optimisation and Evolutionary Operation, *Technometrics*, 1962, 4, pp. 441-446.
15. Nelder, J.A., Mead, R. A simplex method for function minimization, *Comput. J.*, 1965, 7, pp. 308-313.
16. Torczon, V. J. *Multi-Directional Search: A Direct Search Algorithm for Parallel Machines*, PhD Thesis, Rice University, 1989.
17. Hecht-Nielsen R., *Neurocomputing*. Addison-Wesley, 1991.
18. Hertz J., Krogh A., Palmer R.G., *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991.
19. Hagan, M. T., Demuth, H. B., and Beale, M. H., *Neural Network Design*, Boston: PWS Pub., 1996.
20. Beale, R. and Jackson, T., *Neural Computing: An Introduction*, Adam Hilger, 1990.
21. Grefenstette, J., Incorporating Problem-Specific Knowledge into Genetic Algorithms, *Genetic Algorithms and Simulated Annealing*, Edited by L.

- Davis, Morgan Kaufmann Publishers, Los Altos, California, pp. 42-60, 1987.
22. Booker, L., *Improving Search in Genetic Algorithms and Simulated Annealing*, Edited by L. Davis, Morgan Kaufmann Publishers, Los Altos, California, pp. 61-73, 1987.
 23. Franchini, M., Use of a Genetic Algorithm Combined with a Local Search Method for the Automatic Calibration of Conceptual Rainfall-Runoff Models, *Hydrological Sciences Journal*, Vol. 41, pp. 21-39, 1996.
 24. Chipperfield, A. J. and Fleming, P. J., Genetic Algorithms in Control Systems Engineering, *IASTED Journal of Computers and Control*, 24, 1996. pp. 50-56.
 25. Wang, Q., The Genetic Algorithm and Its Application to Calibration of Conceptual Rainfall-Runoff Models, *Water Resources Research*, 27, 1991, pp. 2467-2471.
 26. Bos, M., Bos, A. and Van Der Linden, W. E.; Processing of signals from an ion-selective electrode array by a neural network, *Analytica Chimica Acta*, Vol. 233, 1990, pp. 31-39.
 27. Baughman, D. R. and Liu, Y. A.; *Neural Networks in Bioprocessing and Chemical Engineering*, Academic Press, Inc. A Division of Harcourt Brace & Company, San Diego, California, 1995, pp. 218-219.

APPENDIX A

PROGRAM LISTING

```
/////////////////////////////////////////////////////////////////
// Thesis program
// Title:Simple genetic alogrithm with simplex downhill
// local tuning
// Name:Lu, Jianping
// Place:Department of Computer Science,
// Oklahoma State University
// Date: September 1, 1999
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// This is the main program. In this program, you have a couple
//of chices, such as training the neural network, test the neural
//network, and neural network generalization.
/////////////////////////////////////////////////////////////////
#include"layer_2.h"
#include"option_2.h"
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>

void main()
{
    char response;
    srand( (unsigned)time( NULL ) );

    cout<<"You have four options to run this program."<<endl;
    cout<<"Enter D to test diversity with sigle-minimum function."<<endl;
    cout<<"Enter T to train the network."<<endl;
    cout<<"Enter E to test the network."<<endl;
    cout<<"Enter G to run the generalization."<<endl;
    cin>>response;
    response = toupper(response);
    if(response=='D')
        diversity();
    else if(response=='T')
        training();
    else if(response=='E')
        test();
    else if(response=='G')
```

```

        generalization();
    else{
        cout<<"please enter yes or no."<<endl;
        exit(1);
    }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//option_2.h head file
//In this head file, there are four functions, such as
//training, test, generalization. For training function,
//you have three ways to train the neural network, simple
//genetic method, simplex down hill method, and simple
//genetic method with simplex downhill.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//=====//
//function prototypes
//=====//

void diversity();
void training();
void test();
void generalization();

void diversity()
{
    char what, response;
    int i, points, iterations;
    ofstream outfile;
    outfile.open("dout.dat", ios::out);
    network::get_layer_info('D');
    cout<<"How many points do you want?"<<endl;
    cin>>points;
    GD myGD(points, 'D');
    cout<<"How many iterations do you want?"<<endl;
    cin>>iterations;
    cout<<"Enter 'N' for not using diversity;"<<endl;
    cout<<"Enter 'I' for using diversity;"<<endl;
    cin>>what;
    response = toupper(what);
    switch(response){
        case 'N':
            cout<<"The program is running, please
                be patient."<<endl;
                for(i=0; i<iterations; i++){
                    myGD.single_genetic();
                    myGD.next_single_generation();
                }
            }
    }
}

```



```

switch(choose)
    case 1:
        outfile<<myGD<<endl;
        cout<<"The program is running, please be
        case '1':
            cout<<"The program is running, please
                be patient."<<endl;
                for(i=0; i<iterations; i++){
                    myGD.single_genetic();
                    myGD.single_competition();
                    myGD.next_single_generation();
                }
                cout<<"When the all points are
                outfile<<myGD<<endl;
                break;
                cout<<"The iteration number is: "
            default:
                cout<<"Enter N or I for the single-minimum
                fuction test."<<endl;
        }
    }

//=====//
//function training
//In this function, you have three methods to choose to
//get your weights for the neural network. The final output
//corresponding to the data in the target file, standard
//deviation. and standard deviation are save in the output
//file called out.dat
//=====//
void training()
{
    int i, pd, pg, points, choose, flag, check, sub_sz;
    long iterations;
    ofstream outfile;
    outfile.open("out.dat", ios::out);
    network::get_layer_info("T");
    network::get_input_data();
    network::get_output_data();
    cout<<"How many points do you want?"<<endl;
    cin>>points;
    GD myGD(points);
    cout<<"How many iterations do you want?"<<endl;
    cin>>iterations;
    cout<<"What algorithm do you want to use?"<<endl;
    cout<<"Enter 1 for pure genetic algorithm;"<<endl;
    cout<<"Enter 2 for modified simple genetic algorithm;"<<endl;
    cout<<"Enter 3 for modified simplex downhill algorithm;"<<endl;
    cout<<"Enter 4 for simple genetic with simplex downhill."<<endl;
    cin>>choose;
}

```

```

switch(choose){
    case 1:
        cout<<"The program is running, please be
            patient."<<endl;
        for(i=0; i<iterations; i++){
            myGD.genetic(points);
            myGD.next_generation();
            check = myGD.examine_points();
            if(check==1){
                cout<<"When the all points are
                    the same,"<<endl;
                cout<<"the iteration number is: "
                    <<i+1<<endl;
                break;
            }
        }
        outfile<<myGD<<endl;
        break;
    case 2:
        cout<<"The program is running, please be
            patient."<<endl;
        for(i=0; i<iterations; i++){
            myGD.genetic(points);
            myGD.competition();
            myGD.next_generation();
            check = myGD.examine_points();
            if(check==1){
                cout<<"When the all points are
                    the same,"<<endl;
                cout<<"the iteration number is: "
                    <<i+1<<endl;
                break;
            }
        }
        outfile<<myGD<<endl;
        break;
    case 3:
        cout<<"Now, you chose simplex downhill
            method,"<<endl;
        cout<<"the points you chose is "<<points
            <<endl;
        cout<<"you need to choose a sub_size which
            must be less"<<endl;
        cout<<"than the number of points."<<endl;
        cin>>sub_sz;
        myGD.set_sub_size(sub_sz);
        cout<<"The program is running, please be
            patient."<<endl;

```

```

flag = 1;
for(i=0; i<iterations; i++){
    myGD.downhill(points, flag);
    myGD.competition();
    myGD.next_generation();
    flag = 0;
    check = myGD.examine_points();
    if(check==1){
        cout<<"When the all points are the
                same,"<<endl;
        cout<<"the iteration number is: "
                <<i+1<<endl;
        break;
    }
}
outfile<<myGD<<endl;
break;

```

```

case 4: cout<<"Now, you chose genetic with simplex
        downhill method,"<<endl;
        cout<<"the points you chose is "<<points
        <<endl;
        cout<<"you need to choose a sub_size which
        must be less"<<endl;
        cout<<"than the number of points."<<endl;
        cin>>sub_sz;
        myGD.set_sub_size(sub_sz);
        cout<<"The program is running, please be
        patient."<<endl;
        flag = 1;
        for(i=0; i<iterations; i++){
            pd = points * i/iterations;
            pg = points - pd;
            myGD.genetic(pg);
            //when the pd>=1, downhill starts to work
            if(pd>=1){
                // flag = 1;
                myGD.downhill(pd, flag);
                flag = 0;
            }
            myGD.competition();
            myGD.next_generation();
            check = myGD.examine_points();
            if(check==1){
                cout<<"When the all points are the
                        same,"<<endl;

```



```

ofstream outfile;
char filename1[50], filename2[50];
double max_input, min_input, max_output, min_output;
outfile.open("train_out.dat", ios::out);
cout<<"Enter your input file:"<<endl;
cin>>filename1;
cout<<"Enter your weight file:"<<endl;
cin>>filename2;

network::get_layer_info('G');
cout<<"Enter maximum input, minimum input, maximum
output, minimum output"<<endl;
cout<<"(those data reflect the data when you trained
your network)"<<endl;
cout<<"in this way: max_in min_in max_out, min_out."<<endl;
cin>>max_input>>min_input>>max_output>>min_output;
network::get_input_data();

network net;
net.set_train_weights();
net.calc_out();
outfile<<net;
outfile.close();
}

```

```

/////////////////////////////////////////////////////////////////
//layer_2.h head file
//In this head file, there are three classes, network,
//GD(genetic downhill), and regression. Their functions
//will be introduced when they are implemented, is used
//respectively.
/////////////////////////////////////////////////////////////////
#include<iostream.h>
#include<stdlib.h>
#include<fstream.h>
#include<time.h>
#include<math.h>
//=====//
//Global function random_weights. This function is used
//to generate a random number in a ceratin range when
//it is called.
//=====//
double random_weights()
{
    double number;
    int i;
    //random_weights will return a doubling point
    //value between -20 and 10

    //generate a random number subject to uniform (0, 1)
    //distribution
    for(i=0; i<10; i++)
        number = double (rand()/32767.0);
    //change the random number to a value between -20 to 10
    number = -20 + number * (10 - (-20));

    return number;
}

//=====//
//Global function random_geberator. This function is used
//to generate a random number between 0 and 1 when it is
//called.
//=====//
double random_generator()
{
    int i;
    double number;

    for(i=0; i<10; i++)
        number = double (rand()/32767.0);
}

```

```

    return number;
}
//=====
//Global function segmoid function. this function is used
//to change the input entering a neuron to the ouput of
//the neural in the range of 0 and 1.
//=====
double segmoid(double act)
{
    return 1/(1 + exp(-act));
}
//=====
//class network
//=====
class network
{
private:
    static int layer[5]; //store neurons in the diffrent layer
    static int number_of_layers;
    static double input_data[10][80];
    //nomalized input data array
    static double nom_input_data[10][80];
    static double target_data[10][80];
    //nomalized output data array
    static double nom_target_data[10][80];
    static int num_input_data;
    static int num_target_data;
    static char way;
    static int num_weights;
    double std_dev; //standard deviation
    int num_output_data;
    //store the output of the neural network
    double final_out[10][47];
    //store the weights of the neural network
    double weights[80];
    friend class GD;
public:
    network(); //constructor
    ~network(); //destructor
    static void get_layer_info(char c);
    static void get_input_data();
    static void get_output_data();
    void randomize_weights();
    void calc_out();
}

```

```

void single_calc_out();
void get_first_layer_input(double *temp, int nm);
void calc_temp_out(double *temp1, double *temp2,
    int nr1, int nr2,
    int &count, int last);
void calc_error();
int get_num_weights();
int get_last_layer_nodes();
int get_num_target_data();
static void get_test_input();
static void get_test_output();
void set_train_weights();
friend ostream& operator<<(ostream& myout, network& nt);
};

//=====
/
// Implementation of the constructor
//=====
network::network()
{
    std_dev = 0.0;
}

//=====
/
//Initialization of all class variables.
//=====
/
int network::layer[5];
int network::number_of_layers = 0;
double network::input_data[10][80];
double network::nom_input_data[10][80];
double network::target_data[10][80];
double network::nom_target_data[10][80];
char network::way;
int network::num_input_data = 0;
int network::num_target_data = 0;
int network::num_weights = 0;

//=====
//Implementation of function get_layer_info.

```



```

//=====//
void network::get_layer_info(char c) //=====//
{ //=====//
    int i; //=====//
    //-----//
    //Get layer sizes for the network
    //-----//
    //-----//
    cout<<"Please enter in the number of layers for your
    network."<<endl;
    cout<<"You can have a mininum of 3 to a maximum of
    5."<<endl;
    cout<<"3 implies 1 hidden layer; 5 implies 3 hidden
    layers:"<<endl;

    cin>>number_of_layers;

    cout<<"Enter in the layer sizes separated by spaces."<<endl;
    cout<<"For a network with 3 neurons in the input layer,"<<endl;
    cout<<"2 neurons in a hidden layer, and 4 neurons in the"<<endl;
    cout<<"output layer, you would enter: 3 2 4."<<endl;
    cout<<"You can have up to 3 hidden layers, for five maximum
    entries:"<<endl;

    for(i=0; i<number_of_layers; i++)
        cin>>layer[i];

    //-----//
    //size of layers:
    //      input_layers      layer_size[0]
    //      output_layers      layer_size[number_of_layer-1]
    //      middle_layers      layer_size[1]
    //      optional:layer_size[number_of_layers-3]
    //      optional:layer_size[number_of_layers-2]
    //-----//

    for(i=1; i<number_of_layers; i++)
        num_weights += layer[i-1] * layer[i] + layer[i];

    //two additional weights, one for objective function value,
    //last one for probability
    num_weights = num_weights + 2;
    //assign the way of solving the problem
    way = c;
}

```

```

//=====
//Implementation of function get_input_data.
//=====
void network::get_input_data()
{
    char filename[50];
    double data, min_in, max_in;
    int i, j;
    ifstream infile;

    //get input data from individual file
    for(i=0; i<layer[0]; i++){
        cout<<"Input your input file "<<i+1<<" name:"<<endl;
        cin>>filename;
        infile.open(filename, ios::in|ios::nocreate);
        j=0;
        infile>>data;
        while(!infile.eof()){
            input_data[i][j] = data;
            j++;
            infile>>data;
        }
        num_input_data = j;
        infile.close();
    }

    //normalize the input data
    for(i=0; i<layer[0]; i++){
        min_in = input_data[i][0];
        max_in = input_data[i][0];
        for(j=0; j<num_input_data; j++){
            if(input_data[i][j]<min_in)
                min_in = input_data[i][j];
            if(input_data[i][j]>max_in)
                max_in = input_data[i][j];
        }
        input_data[i][num_input_data] = min_in;
        input_data[i][num_input_data+1] = max_in;
        for(j=0; j<num_input_data; j++)
            nom_input_data[i][j] = 1.0/(max_in-min_in)*
                (input_data[i][j]-min_in);
    }
}

//=====

```

```

//This function is to get input test data.
//=====//
void network::get_test_input()
{
    //get output
    char filename[50];
    double data, min_in, max_in;
    int i, j;
    ifstream infile;

    //get input data from individual file
    for(i=0; i<layer[0]; i++){
        cout<<"Input your test input file "<<i+1<<
            " name:"<<endl;
        cin>>filename;
        infile.open(filename, ios::in|ios::nocreate);
        j=0;
        infile>>data;
        while(!infile.eof()){
            input_data[i][j] = data;
            j++;
            infile>>data;
        }
        num_input_data = j;
        infile.close();
        cout<<"Input max_in during the corresponding input
            training file:"<<endl;
        cin>>max_in;
        cout<<"Input min_in during the corresponding input
            training file:"<<endl;
        cin>>min_in;
        input_data[i][num_input_data] = min_in;
        input_data[i][num_input_data+1] = max_in;
        for(j=0; j<num_input_data; j++)
            nom_input_data[i][j] = 1.0/(max_in-min_in)*
                (input_data[i][j]-min_in);
    }
}

//=====//
//This function is to get input data for
//training the neural network.
//=====//
void network::get_output_data()
{
    char filename[50];
    double data, min_out, max_out;

```

```

int i, j;
ifstream infile;
//get output data from individual file
for(i=0; i<layer[number_of_layers-1]; i++){
    cout<<"Input your output file "<<i+1<<" name:"<<endl;
    cin>>filename;
    infile.open(filename, ios::in|ios::nocreate);
    j=0;
    infile>>data;
    while(!infile.eof()){
        target_data[i][j] = data;
        j++;
        infile>>data;
    }
    num_target_data = j;
    infile.close();
}

//normalize the output data
for(i=0; i<layer[0]; i++){
    min_out = target_data[i][0];
    max_out = target_data[i][0];
    for(j=0; j<num_target_data; j++){
        if(target_data[i][j]<min_out)
            min_out = target_data[i][j];
        if(target_data[i][j]>max_out)
            max_out = target_data[i][j];
    }
    target_data[i][num_target_data] = min_out;
    target_data[i][num_target_data+1] = max_out;
    for(j=0; j<num_target_data; j++)
        nom_target_data[i][j] = 1.0/(max_out-min_out)*
            (target_data[i][j]-min_out);
}
}

//=====//
//This function is to get output data for test the
//neural network.
//=====//
void network::get_test_output()
{
    char filename[50];
    double data, min_out, max_out;
    int i, j;

```

```

ifstream infile;

//get output data from individual file
for(i=0; i<layer[number_of_layers-1]; i++){
    cout<<"Input your output file "<<i+1<<
        " name:"<<endl;
    cin>>filename;
    infile.open(filename, ios::in|ios::nocreate);
    j=0;
    infile>>data;
    while(!infile.eof()){
        target_data[i][j] = data;
        j++;
        infile>>data;
    }
    num_target_data = j;
    infile.close();
    cout<<"Input max_out during the corresponding output
        training file:"<<endl;
    cin>>max_out;
    cout<<"Input min_out during the corresponding output
        training file:"<<endl;
    cin>>min_out;
    target_data[i][num_target_data] = min_out;
    target_data[i][num_target_data+1] = max_out;
    for(j=0; j<num_target_data; j++)
        nom_target_data[i][j] = 1.0/(max_out-min_out)*
            (target_data[i][j]-min_out);
}
}

//=====//
//This function is to get number of the target data.
//=====//
int network::get_num_target_data()
{
    return num_target_data;
}

//=====//
//This function is to get how many nodes in the output layer
//of the neural network.
//=====//
int network::get_last_layer_nodes()
{
    return layer[number_of_layers-1];
}

```

```

}
//=====//
void network::calc_out()
//=====//
//Implementation of function set_train_weights.
//This function is only used
//when the user asks test or generalize the neural network.
//=====//
void network::set_train_weights()
{
    char filename[50];
    ifstream infile;
    int i;
    double weight;
    i = 0;

    cout<<"Input your weight file name:"<<endl;
    cin>>filename;
    infile.open(filename, ios::in|ios::nocreate);

    infile>>weight;
    while(!infile.eof()){
        weights[i] = weight;
        infile>>weight;
        i++;
    }
    infile.close();
}

//=====//
//Implementation of function randomize_weights.
//This function is used to fill the
//random weights for the network.
//=====//
void network::randomize_weights()
{
    int i;

    for(i=0; i<num_weights-2; i++)
        weights[i] = random_weights();
}

//=====//
//Implementation of function calc_out.
//This function is used to get the output
//the neurons in the output layer.

```

```

//=====//
void network::calc_out()
{
    int h, i, j, k, m, n, weight_count, last_layer;
    n = 0;
    i = 0;
    h = 0;
    k = 0;

    double temp1_out[100], temp2_out[100];
    while(k<num_input_data){
        weight_count = 0;
        get_first_layer_input(temp1_out, k);
        for(j=1; j<number_of_layers; j++){
            last_layer = j;
            calc_temp_out(temp1_out, temp2_out, layer[j-1],
                layer[j], weight_count, last_layer);
            for(m=0; m<layer[j]; m++)
                temp1_out[m] = temp2_out[m];
        }
        i++;

        for(n=0; n<layer[number_of_layers-1]; n++){
            final_out[n][k] = temp1_out[n];
        }
        k++;
    }
}

```

```

//=====//
//Implementation of function get_first_layer_input.
//=====//

```

```

void network::get_first_layer_input(double *temp, int nm)
{
    int i;
    for(i=0; i<layer[0]; i++){
        temp[i] = nom_input_data[i][nm];
    }
}

```

```

//=====//
//Implementation of function calc_temp_out.
//This function is used to calculate

```

```

// the output of the neurons in the hidden layers.
//=====//
void network::calc_temp_out(double *temp1, double *temp2,
                           int nr1, int nr2, int &count,
                           int last)
{
    int i, j;
    double value;

    for(i=0; i<nr2; i++){
        value = 0.0;
        for(j=0; j<nr1; j++){
            value += temp1[j]*weights[count];
            (count)++;
        }
        value = value + weights[count];
        (count)++;
        if(last!=number_of_layers-1)
            temp2[i] = sigmoid(value);
        else
            temp2[i] = value;
    }
}

//=====//
//Implementation of function calc_error.
//this function is used to calculate
//the standard deviation of the output the
//neural network related to the target data.
//=====//
void network::calc_error()
{
    int i, j;
    double sum;
    sum = 0.0;
    for(i=0; i<layer[number_of_layers-1]; i++)
        for(j=0; j<num_target_data; j++)
            sum += (nom_target_data[i][j]-final_out[i][j])*
                (nom_target_data[i][j]-final_out[i][j]);
    std_dev = sqrt(sum);
    weights[num_weights-2] = std_dev;
}

void network::single_calc_out()
{
    int i;

```



```

    double value;
    value = 0.0;
    for(i=0; i<num_weights-2; i++)
        value += weights[i] * weights[i];
//    std_dev = sqrt(value/double (num_weights-3));
    weights[num_weights-2] = value;
}

//=====//
//Implementation of function get_num_weights.
//This function is used to get the total number
//of weights in the neural network.
//=====//
int network::get_num_weights()
{
    return num_weights;
}

//=====//
//Implementation of friend function to print results.
//=====//
ofstream& operator<<(ofstream& myout, network& nt)
{
    int i, j;
    double data;

    myout<<"The final outputs are:"<<endl;
    if(nt.way!='D')
        for(i=0; i<nt.get_last_layer_nodes(); i++){
            for(j=0; j<nt.num_target_data; j++){
                //denormliazation
                data = nt.final_out[i][j]*
                    (nt.target_data[i][nt.get_num_target_data()+1]-
                     nt.target_data[i][nt.get_num_target_data()])+
                    nt.target_data[i][nt.get_num_target_data()];
                myout<<data<<endl;
            }
            myout<<endl;
        }
    myout<<endl;
    if(nt.way=='T'||nt.way=='D'){
        myout<<"the standard deviation is: "<<
            nt.weights[nt.num_weights-2]<<endl;
        myout<<"the weights are: "<<endl;
        for(i=0; i<nt.get_num_weights()-2; i++)
            myout<<nt.weights[i]<<" ";
    }
}

```

```

        myout<<endl;
    }
    if(nt.way=='E'){
        nt.calc_error();
        myout<<"the standard deviation is: "<<
            nt.weights[nt.num_weights-2]<<endl;
    }
    return myout;
}

//=====//
//destructor of network.
//=====//
network::~network()
{
}

//=====//
//class GD(genetic downhill method
//=====//
class GD
{
private:
    network net[100];
    network new_net[100];
    int num_of_points;
    char C;
    int sub_size;
    network two_points[2]; //for genetic, to choose two parents
    network cross;
    //for downhill, to choose a subcommunity
    network first_s_points[10];
    //for downhill, to choose a subcommunity
    network second_s_points[10];
public:
    GD(int pts);//constructor
    GD(int pts, char c);//constructor
    ~GD() {}; //destructor
    void genetic(int PG);
    void single_genetic();
    void downhill();
    void heapsort(network arr[], int size);
    void heapify(network arr[], int pos, int size);
    void assign_probability();
    int get_index(double rd);
    void select_two_points();
}

```

```

} void crossover();
void mutation();
//void select_s_points(int Flag);
//void construct_subcommunity(int Flag);
//void compute_centroid(int dn);
void GD::downhill(int PD, int Flag);
void competition();
void single_competition();
double distance(network nt[], int index);
void next_generation();
void next_single_generation();
int examine_points();
void set_sub_size(int sub_siz);
friend ostream& operator<<(ostream& out, GD& gd);
};

//=====//
//Implementation of constructor
//=====//
GD::GD(int pts)
{
    int i;
    num_of_points = pts;
    for(i=0; i<num_of_points; i++){
        net[i].randomize_weights();
        net[i].calc_out();
        net[i].calc_error();
    }
    heapsort(net,num_of_points);
    assign_probability();
}

//=====//
//Implementation of constructor
//=====//
GD::GD(int pts, char c)
{
    int i;
    num_of_points = pts;
    for(i=0; i<num_of_points; i++){
        net[i].randomize_weights();
        net[i].single_calc_out();
    }
    heapsort(net,num_of_points);
    assign_probability();
}

```

```

}
//Implementation of single genetic.
//Implementation of function heapsort, to sort the
//points in ascending order according to the
//objective function values.
void GD::single_genetic()
{
    genetic(num_of_points);
}

//Implementation of function heapsort, to sort the
//points in ascending order according to the
//objective function values.
void GD::heapsort(network arr[], int size)
{
    network x, temp;

    int i, j;
    for(i=(size-1)/2; i>=0; i--)
        heapify(arr, i, size);

    for(i=size-1; i>0; i--){
        for(j=0; j<temp.get_num_weights()-1; j++){
            x.weights[j] = arr[0].weights[j];
            arr[0].weights[j] = arr[i].weights[j];
            arr[i].weights[j] = x.weights[j];
        }
        heapify(arr, 0, i);
    }
}

//Implementation of heapify function.
void GD::heapify(network arr[], int pos, int size)
{
    int j, l, r, k, largest;
    network x, temp;

    j = pos;
    while(j<size-1){
        l = 2*j;
        r = 2*j+1;

```

```

        if(l<=size-1){
            if(arr[l].weights[temp.get_num_weights()-2]>
                arr[j].weights[temp.get_num_weights()-2])
                largest = l;
            else
                largest = j;
        }
        if(r<=size-1){
            if(arr[r].weights[temp.get_num_weights()-2]>
                arr[largest].weights[temp.get_num_weights()-2])
                largest = r;
        }
        if(largest!=j){
            for(k=0; k<temp.get_num_weights()-1; k++){
                x.weights[k] = arr[j].weights[k];
                arr[j].weights[k] = arr[largest].weights[k];
                arr[largest].weights[k] = x.weights[k];
            }
            j = largest;
        }
        else
            break;
    } //end_while
}

//=====//
//Implementation of function get_index.
//This function is used to find a point
//index by the probability assigned to
//it when a random number is given.
//=====//
int GD::get_index(double rd)
{
    network temp;

    double start, end;
    int i, target_index;
    start = end = 0.0;
    if(rd<net[0].weights[temp.get_num_weights()-1])
        target_index = 0;
    for(i=1; i<num_of_points; i++){
        start += net[i-1].weights[temp.get_num_weights()-1];
        end = start+net[i].weights[temp.get_num_weights()-1];
        if(start<rd && rd<=end){
            target_index = i;
        }
    }
}

```

```

    } flag = 1;
    return target_index;
}

//=====
//Implementation of function assign_probability.
//This function is used to assign a probability
//number to a point according to its objective
//function value. The smaller the objective
//function value is, the greater the probability
//number is.
//=====
void GD::assign_probability()
{
    network temp;
    int i;
    double c, pm, pb, p;
    c = 0.5;
    pm = 0.5/double(num_of_points);
    pb = (2-c)/double(num_of_points);

    for(i=num_of_points; i>0; i--){
        p = pm+(double(i-1)/double(num_of_points-1))*(pb-pm);
        //put the probability in the last row of the weight array
        net[num_of_points-i].weights[temp.get_num_weights()-1] = p;
    }
}

//=====
//Implementation of function set_sub_size.
//this function is for downhill method.
//=====
void GD::set_sub_size(int sub_siz)
{
    sub_size = sub_siz;
}

//=====
//Implementation of select_two_points.
//This function is for genetic method.
//=====
void GD::select_two_points()
{
    network temp;
    int i, j, index, flag;
    double rd_num;

```

```

flag = 1; num = random_generator()
while(flag == 1){ < 0.50)
    for(i=0; i<2; i++){
        rd_num = random_generator();
        index = get_index(rd_num);
        for(j=0; j<temp.get_num_weights()-2; j++){
            two_points[i].weights[j] = net[index].weights[j];
        }
        //check whether the points are the same
        for(i=0; i<temp.get_num_weights()-2; i++){
            if(two_points[0].weights[i] != two_points[1].weights[i]){
                flag = 0;
                break;
            }
        }
    }
}

//=====//
//Implementaion of function crossover,
//This function is for genetic method.
//=====//
void GD::crossover()
{
    network temp;
    int i;
    double rd_num;

    for(i=0; i<temp.get_num_weights()-2; i++){
        rd_num = random_generator();
        if(rd_num >= 0.5)
            cross.weights[i] = two_points[0].weights[i];
        else
            cross.weights[i] = two_points[1].weights[i];
    }
}

//=====//
//Implementation of function mutation.
//This function is for genetic method.
//=====//
void GD::mutation()
{
    network temp;
    double rd_num;
    int i;
    for(i=0; i<temp.get_num_weights()-2; i++){

```

```

        rd_num = random_generator();
        if(rd_num < 0.50)
            cross.weights[i] = random_weights();
    }
}

//=====//
//Implementation of function genetic.
//In this function, the genetic way to
//create a new population is carried out.
//=====//
void GD::genetic(int PG)
{
    network temp;
    int i, j;
    for(i=0; i<PG; i++){
        select_two_points();
        crossover();
        mutation();
        //copy the point after mutation into weights
        for(j=0; j<temp.get_num_weights()-2; j++)
            new_net[i].weights[j] = cross.weights[j];
    }
}

//=====//
//Implementation of function select_s_points.
//This function is for downhill method.
//=====//
void GD::select_s_points(int Flag)
{
    network temp;
    int i, j, k, same;
    int index;
    double rd_num;
    same = 1;

    if(Flag==1){
        for(i=0; i<sub_size; i++){
            rd_num = random_generator();
            index = get_index(rd_num);
            for(j=0; j<temp.get_num_weights()-2; j++)
                first_s_points[i].weights[j] = net[index].weights[j];
        }
    }
    //if not first time to create three points

```



```

else{
    while(same==1){
        for(i=0; i<sub_size; i++){
            rd_num = random_generator();
            index = get_index(rd_num);
            for(j=0; j<temp.get_num_weights()-2; j++)
                second_s_points[i].weights[j] =
                    net[index].weights[j];
        }
        //now to check the second_s_points to see whether it is
        //identical to the first_s_points
        for(i=0; i<sub_size; i++){
            for(k=0; k<sub_size; k++){
                for(j=0; j<temp.get_num_weights()-2; j++){
                    if(first_s_points[i].weights[j]!=
                       second_s_points[k].weights[j]){
                        same = 0;
                        break;//break the first for
                    }
                }
                //end_j_for
                if(same==0)
                    break;//break the second for
            }
            //end_k_for
            if(same==0)
                break;
        }
        //end_i_for
    }
    //end_while
    //now copy the second_s_points into the first_s_points
    for(i=0; i<sub_size; i++)
        for(j=0; j<temp.get_num_weights()-2; j++)
            first_s_points[i].weights[j] =
                second_s_points[i].weights[j];
}
}

//=====//
//Implementation of construct_subcommunity.//
//=====//
void GD::construct_subcommunity(int Flag)
{
    int i;
    select_s_points(Flag);
    for(i=0; i<sub_size; i++){
        first_s_points[i].calc_out();
        first_s_points[i].calc_error();
    }
}

```

```

}
//=====
//Implementation of function compute_centroid.
//This function is used to create
//a new population of points.
//=====
void GD::compute_centroid(int dn)
{
    network temp1, temp2;
    temp1.randomize_weights();
    temp2.randomize_weights();
    int i, j;
    double value;

    heapsort(first_s_points, sub_size);
    //compute the centroid of them without including the worst point
    for(i=0; i<temp1.get_num_weights()-2; i++){
        value = 0.0;
        for(j=0; j<sub_size-1; j++)
            value += first_s_points[j].weights[i];
        value = value/double(sub_size-1);
        temp1.weights[i] = value;
    }
    temp1.calc_out();
    temp1.calc_error();

    //construct a new point by reflecting the worst point through the
    //centroid point  $y(ij) = 2x(ic) - x(is)$ 
    for(i=0; i<temp1.get_num_weights()-2; i++)
        temp2.weights[i] = 2*temp1.weights[i] -
            first_s_points[sub_size-1].weights[i];
    temp2.calc_out();
    temp2.calc_error();

    if(temp2.weights[temp1.get_num_weights()-2]<
        first_s_points[sub_size-1].weights[temp1.get_num_weights()-2]){
        for(i=0; i<temp1.get_num_weights()-2; i++)
            new_net[dn].weights[i] = temp2.weights[i];
    }
    else if(temp1.weights[temp1.get_num_weights()-2]<
        first_s_points[sub_size-1].weights[temp1.get_num_weights()-2]){
        for(i=0; i<temp1.get_num_weights()-2; i++)
            new_net[dn].weights[i] = temp1.weights[i];
    }
    else{

```

```

        for(i=0; i<temp1.get_num_weights()-2; i++){
            new_net[dn].weights[i] = random_weights();
        }
    }

//=====//
//Implementation of function downhill.
//This function is used to perform the
//downhill method.
//=====//
void GD::downhill(int PD, int Flag)
{
    int i;
    for(i=(num_of_points-PD); i<num_of_points; i++){
        construct_subcommunity(Flag);
        compute_centroid(i);
    }
}

//=====//
//Implementation of function competition,
//After the execution of the this function,
//the best-so-far point is always kept for next generation.
//=====//
void GD::competition()
{
    network temp;
    int i, j;
    double rd_num;
    for(i=0; i<num_of_points; i++){
        new_net[i].calc_out();
        new_net[i].calc_error();
    }
    heapsort(new_net, num_of_points);
    //check the best point in the new_net and the best point
    //in the net
    if(new_net[0].weights[temp.get_num_weights()-2]>
        net[0].weights[temp.get_num_weights()-2]){
//replace the point in the new_net by the corresponding point int net
        for(i=0; i<temp.get_num_weights()-1; i++)
            new_net[num_of_points-1].weights[i] = net[0].weights[i];
        heapsort(new_net, num_of_points);
    }
    //check other points
    for(i=1; i<num_of_points; i++){

```

```

//in the if((new_net[i].weights[temp.get_num_weights()-2]<
//new_net[0].weights[temp.get_num_weights()-2])&&
//net[0].weights[temp.get_num_weights()-2]>distance(new_net, i)>distance(net, i)){
//replace the point //keep the new point
}
else if((distance(net, i)>distance(new_net, i))&&
(new_net[i].weights[temp.get_num_weights()-2]<
new_net[0].weights[temp.get_num_weights()-2])){
//replace the new point with the corresponding point
for(j=0; j<temp.get_num_weights()-1; j++){
new_net[i].weights[j] = net[i].weights[j];
heapsort(new_net, num_of_points);
}
else if(distance(new_net, i)*
net[i].weights[temp.get_num_weights()-2]>
distance(net, i)*
new_net[0].weights[temp.get_num_weights()-2]){
}
else{
rd_num = random_generator();
if(rd_num>0.5){
for(j=0; j<temp.get_num_weights()-1; j++){
new_net[i].weights[j] = net[i].weights[j];
heapsort(new_net, num_of_points);
}
else
;
}
}
}
}
}

```

```

//=====//
//This function is used for single competition.
//=====//

```

```

void GD::single_competition()
{
network temp;
int i, j;
double rd_num;
for(i=0; i<num_of_points; i++){
new_net[i].single_calc_out();
}
heapsort(new_net, num_of_points);
//check the best point in the new_net and the best point

```

```

//in the net
if(new_net[0].weights[temp.get_num_weights()-2]>
    net[0].weights[temp.get_num_weights()-2]){
//replace the point in the new_net by the corresponding point int net
for(i=0; i<temp.get_num_weights()-1; i++)
    new_net[num_of_points-1].weights[i] = net[0].weights[i];
heapsort(new_net, num_of_points);
}
//check other points
for(i=1; i<num_of_points; i++){
    if(new_net[i].weights[temp.get_num_weights()-2]<
        net[i].weights[temp.get_num_weights()-2]&&
        distance(new_net, i)>distance(net, i)){
        //keep the new point
        ;
    }
    else if(new_net[i].weights[temp.get_num_weights()-2]>
        net[i].weights[temp.get_num_weights()-2]&&
        distance(net, i)>distance(new_net, i)){
        //replace the new point with the corresponding point
        for(j=0; j<temp.get_num_weights()-1; j++)
            new_net[i].weights[j] = net[i].weights[j];
        heapsort(new_net, num_of_points);
    }
    else if(new_net[i].weights[temp.get_num_weights()-2]<
        net[i].weights[temp.get_num_weights()-2]&&
        distance(new_net, i)>distance(net, i)){
        //keep the new point
        ;
    }
    else if(distance(new_net, i)*
        net[i].weights[temp.get_num_weights()-2]>
        distance(net, i)*
        new_net[i].weights[temp.get_num_weights()-2]){
        ;
    }
    else{
        rd_num = random_generator();
        if(rd_num>0.5){
            for(j=0; j<temp.get_num_weights()-1; j++)
                new_net[i].weights[j] = net[i].weights[j];
            heapsort(new_net, num_of_points);
        }
        else
            ;
    }
}

```

```

    }
}
//=====//
//Implementation of function distance.
//this function is used to calculate
//the distance between two points.
//=====//
double GD::distance(network nt[], int index)
{
    network temp;
    double d;
    int i;
    d = 0.0;
    for(i=0; i<temp.get_num_weights()-2; i++)
        d += (nt[index].weights[i] - new_net[0].weights[i])*
            (nt[index].weights[i]-
            new_net[0].weights[i]);
    return sqrt(d);
}

//=====//
//Implementation of function next_generation.
//=====//
void GD::next_generation()
{
    network temp;
    int i, j;
    for(i=0; i<num_of_points; i++){
        new_net[i].calc_out();
        new_net[i].calc_error();
    }
    heapsort(new_net, num_of_points);
    //copy new_net into net, ready for next generation
    for(i=0; i<num_of_points; i++){
        for(j=0; j<temp.get_num_weights()-1; j++)
            net[i].weights[j] = new_net[i].weights[j];
    }

    for(i=0; i<num_of_points; i++){
        net[i].calc_out();
        net[i].calc_error();
    }
    heapsort(net, num_of_points);
    assign_probability();
}
}

```

```

//=====//
//This function is used for single next generation.
//=====//
void GD::next_single_generation()
{
    network temp;
    int i, j;
    for(i=0; i<num_of_points; i++){
        new_net[i].single_calc_out();
    }
    heapsort(new_net, num_of_points);
    //copy new_net into net, ready for next generation
    for(i=0; i<num_of_points; i++){
        for(j=0; j<temp.get_num_weights()-1; j++){
            net[i].weights[j] = new_net[i].weights[j];
        }

        for(i=0; i<num_of_points; i++){
            net[i].single_calc_out();
        }
        heapsort(net, num_of_points);
        assign_probability();
    }
}

```

```

//=====//
//Implementation of examine_points. If the all
//corresponding weights in the allpoints are
//the same, thus the global minumumis approached,
//therefore the program should be stopped.
//=====//
int GD::examine_points()
{
    network temp;
    int i, j, repeat;
    int total_weights;
    repeat = 0;
    total_weights = 0;
    total_weights = (num_of_points-1) * (temp.get_num_weights()-2);
    for(i=1; i<num_of_points; i++)
        for(j=0; j<temp.get_num_weights()-2; j++)
            if(net[0].weights[j]==net[i].weights[j])
                repeat++;
    if(repeat==total_weights)
        return 1;
}

```

```
else
    return 0;
}

//=====//
//Implementation of friend function. To print the results.
//=====//
ofstream& operator<<(ofstream& out, GD& gd)
{
    out<<gd.net[0];
    return out;
}
```

GIFT

GIFT

VITA

Jianping Lu

Candidate for the Degree of

Master of Science

Thesis: SIMPLE GENETIC ALGORITHM WITH SIMPLEX LOCAL TUNING FOR EFFICIENT GLOBAL OPTIMIZATION

Major field: Computer Science

Biographical:

Education: March 1978 to January 1981, Changcun University of Science and Technology, P. R. China; received Bachelor of Science degree in Chemistry. September, 1983 to June, 1986, Beijing University of Science and Technology, P. R. China; received Master of Science degree in Chemistry. August, 1992 to August, 1997, Texas Tech University, Lubbock, Texas, USA; received Ph.D. in Chemistry. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May, 2000.

Experience: July, 1986 to April, 1992, employed as a lecturer in Zhejiang University, Hangzhou, Zhejiang, P. R. China.