1170212

GENETIC ALGORITHM WITH CHEMOTAXIS TUNABLE LOCAL SEARCHING OPTIMIZATION

By

WEI LI

Bachelor of Science

Changchun University

Science and Technology

Changchun, China

1982

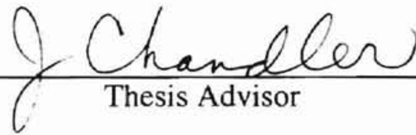
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
In partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2001

GENETIC ALGORITHM WITH CHEMOTAXIS

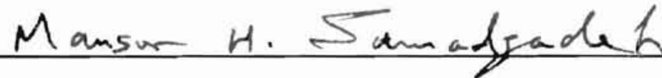
TUNABLE LOCAL SEARCHING

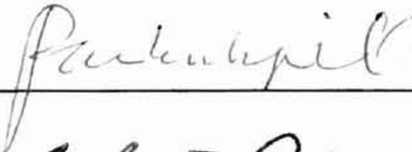
OPTIMIZATION

Thesis Approved:



Thesis Advisor





Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express sincere appreciation to the chairperson of my thesis committee, Dr. Chandler, for his time and effort. He has provided helpful advice and encouragement all through the various stages of the program. I would like to thank the other members of my committee, Dr. Samadzadeh and Dr. Park, for their helpful comments and suggestions. I would like to thank my family, my husband Jianping Lu, my daughter Zhen Lu and my son Jeffrey Lu, for their encouragement of my aspirations; their faith in me never lagged.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION ... 1

1.1 Principle of Genetic Algorithm ... 1

1.2 Principle of Chemotaxis ... 10

1.3 Principle of Simulated Annealing ... 12

1.4 Artificial Neural Networks ... 14

II. METHODOLOGY ... 19

2.1 Comparison of GA and CA ... 20

2.2 The Proposed Algorithm GADCA ... 22

2.3 Case Studies and the Selection of Main Parameters ... 24

III. RESULTS AND DISCUSSION ... 25

3.1 Case 1 ... 25

3.2 Case 2 ... 49

IV. CONCLUSIONS ... 54

REFERENCES ... 56

APPENDIX A – PROGRAM LISTING ... 60

LIST OF TABLES

Table | Page

3-1 Training data for the lubricant viscosity at different temperature and pressure 26

3-2 Comparison of the impact of using genetic diversity guidance for the genetic algorithm (epoch = 100)..........30

3-3 Comparison of the impact of using genetic diversity guidance for the genetic algorithm (epoch = 1000) 32

3-4 Comparison of the impact of using genetic diversity guidance for the genetic algorithm (epoch = 2000)..........34

3-5 Comparison of the genetic algorithm with diversity guidance (GAD) and the genetic algorithm with diversity guidance combined with chemotaxis (GADC)..........36

3-6 Comparison of the genetic algorithm with diversity guidance (GAD), chemotaxis algorithm (CA) and the genetic algorithm with diversity guidance combined with chemotaxis (GADCA).. 38

3-7 Influence of varying the parameters on the performance of GADCA 40

3-8 The influence of varying the number of neurons in the hidden-layer to the performance of GADCA..........42

3-9 Training result for the lubricant viscosity at different temperature and pressure in Table 3-1..........43

3-10 Test data for the lubricant viscosity at different temperature and pressure 46

3-11 Testing result for the test data set in Table 3-10..........46

3-12 Generalization data set 47

3-13 Generalization results for the data in Table 3-12..........48

3-14 Vapor pressure of water 50

3-15 Training results for the data in Table 3-14......52

3-16 Test data for water vapor at different temperatures......53

3-17 Testing results for the data in Table 3-16 53

LIST OF FIGURES

Figure Page

1-1 Initial population 4

1-2 Population after reproduction 6

1-3 Crossover 6

1-4 Population after crossover 7

1-5 Final population after mutation 8

3-1 Effect of the variation of population size on the objective function value (error) for pure genetic algorithm and genetic algorithm with diversity guidance, respectively, when the epoch = 100....................32

3-2 Effect of the variation of population size on the objective function value (error) for pure genetic algorithm and genetic algorithm with diversity guidance, respectively, when the epoch = 1000....................33

3-3 Effect of the variation of population size on the objective function value (error) for pure genetic algorithm and genetic algorithm with diversity guidance, respectively, when the epoch = 2000....................35

CHAPTER I INTRODUCTION

One of the most fundamental problems in applied mathematics is that of optimization to maximize or minimize a simple function. Consider as an example the function $f(x) = 10 - (x-2)^3$. By differentiating f with respect to x, setting the derivative equal to zero, and solving for x, one can generate a list of possible local extreme value for the function. While all optimization problems can be viewed as an extension of this example, most, unfortunately, cannot be solved so easily. In more typical applications, the objective function has more than one variable.

Another complication is in the fact that many functions are not so easily defined or differentiated. Not all functions can be written in terms of a mathematical expression, and many complex functions are difficult or impossible to differentiate. In an attempt to find other ways of solving these more realistic, less well-behaved optimization problems, other computational techniques have been investigated, some popular methods that show some promise are the genetic algorithm [1, 2], the chemotaxis algorithm [3], the simplex algorithm [4], the simulated annealing algorithm [5], etc. We are focusing on discussing the principles of genetic algorithms and chemotaxis algorithms, since these two algorithms are implemented in this thesis.

1.1 Principles of Genetic Algorithms

The genetic algorithm (GA) is a combinatorial optimizer that is domain-independent: it is applicable to all functions that can be evaluated. The genetic algorithm requires only two

things: (1) a means of representing possible solutions and (2) an objective function evaluator which is a function that maps a value from the domain of possible solutions to a scalar value. The genetic algorithm starts with a computer-created population of individuals, each representing a point in the search space of a given function. Using an individual's objective function as a measure of how "fit" that individual is within its environment, the genetic algorithm simulates nature's survival of the fittest, essentially forcing the evolution of a nearly optimal creature. This early optimal creature is then the approximate solution to the corresponding optimization problem.

The genetic algorithm has been implemented in various forms since its introduction in the late of 1960s. As its name suggests, the first research done on genetics-based algorithms was not motivated by unsolved optimization problems. Instead, these algorithms were designed as simulations of natural adaptive processes. Most researchers in the young field of adaptation-simulation used models with properties closely resembling natural phenomena. For example, the biological notions of diploid chromosomes and dominance were both frequently mimicked by early algorithms. John Holland [6], a professor at the University of Michigan, was one of the first researchers to carry out a substantial amount of work in the field. He recognized the broad applicability of genetics-based algorithms for optimization purposes, and this insight formed the basis for the modern notion of a genetic algorithm.

Despite its power, the genetic algorithm is both elegant and simple. That such a simple, straightforward routine can accomplish so much is quite unexpected. The genetic algorithm contains only one main data structure: a population of individuals. Each individual, affectionately known as a critter, represents an element within the domain of

the solution space of the optimization problem; i.e., each critter represents a possible solution to the problem. The issue of how to best represent a critter is very complex and has tremendous problem-solving implications. In the simplest genetic algorithm, critters are simple strings of bits (binary digits: ones and zeros). Each string of ones and zeros is called a chromosome; the chromosome of a given critter is the only source for all the information about the corresponding solution. In biological terms, the chromosomal string is the genotype and the solution it represents the phenotype of a particular critter.

Associated with each individual is a fitness value. The value is a numerical quantification of how good a solution to the optimization problem the individual is. Individuals with chromosomal strings representing better solutions have higher fitness value, while lower fitness values are attributed to those whose bit strings represent inferior solutions [7].

It is important to realize that only two elements of the genetic algorithm need to be changed in order to apply the algorithm to a new problem: the representation of the individuals and the objective functions. Consider, for example, one of the most basic test problems the GA is applied to: One Max. The goal in One Max is to maximize the number of occurrences of digit 1 in an arbitrarily long string of bits. As an example, let us assume that strings are eight bits long. The representation of an individual is thus a string of eight ones and zeros: 10110001, for example. Standard GA terminology refers to each bit position as a locus and to the values at the loci as alleles. The set of all symbols which an allele can assume is called the alphabet of the representation. In our examples, the alphabet consists of 0 and 1 [8].

Since the goal in One Max is to maximize the number of 1 bits, we need an objective function evaluator which gives better ratings to individuals with more 1 bits. The obvious choice is the function which assigns as an individual's fitness value the number of ones in its representation; e.g., 10110001 has fitness four, while 00000000 has fitness zero. The goal, then, of our algorithm is to find the individual with fitness value eight: 11111111.

Now that we have a suitable representation and an appropriate function, the construction of the genetic algorithm is almost complete. One of the important parameters of any GA is population size, which is how many critters are maintained at any given time. In our One Max example, we will assume a population size of four; populations are typically much larger, often 20 to 200. Since we intend to have four critters "alive" in the current population at any given time, the GA must create four individuals to form the initial population. In the GA, these initial individuals are merely random bit strings. Thus our initial population might consist of the four individuals in Figure 1-1, where each X_i is a critter in the population.

Critter's String	Fitness	Selection Probability
X_1 = 00101111	5	5/17
X_2 = 00111010	4	4/17
X_3 = 10111011	6	6/17
X_4 = 10000100	2	2/17

Figure 1-1. Initial population.

Common sense tells us that some of the initial individuals probably are going to be better than others. That is, some bit strings will score higher fitness values, meaning they are better solutions to the One Max problem. Analogously, some of the critters in the initial population will be better adapted to their environment. In nature, those individuals that are better adapted are more likely to survive. Survival of the fittest is mirrored in the genetic algorithm through reproduction, one of the three main genetic operators.

The GA thus creates a second generation of individuals. Since the population size must remain constant, however, each new individual must replace an old one. The GA creates a population of new individuals to replace the previous generation; in our example, the GA would create four new individuals. Each new individual will be identical to a certain previous generation. Specifically, the probability of an individual X_k in the first generation reproducing is $f(X_k)/\Sigma f(X_i)$.

We can thus list for each of the individuals in our initial population that individual's fitness value and the probability of it reproducing, shown in Figure 1-1. To continue with our One Max example, we will assume that X_3 reproduces twice, that X_1 and X_2 each reproduce once, and that X_4, the least fit individual, fails to reproduce, thus yielding the new population depicted in Figure 1-2.

The next step of the GA distinguishes it from other domain-independent optimization techniques. In this step, the crossover operator, which is the second main genetic operator, is repeatedly applied to pairs of individuals. Suppose for example that critters one and three are chosen to mate or to be crossed. This would leave critters two and four to be crossed. The process of crossing two individuals involves randomly selecting a locus and then swapping between the two individuals their genetic materials following

that locus. If in our example the crossover point selected for critters one and three were the fourth locus, the resulting strings would be 10111111 and 00101011 shown in Figure 1-3. Likewise, if the sixth locus were selected as the crossover point for X2 and X4, the individuals 10111010 and 00111011 would be formed.

Critter's String	Fitness	Selection Probability
X_1 = 10111011	6	6/21
X_2 = 10111011	6	6/21
X_3 = 00101111	5	5/21
X_4 = 00111010	4	4/21

Figure 1-2. Population after reproduction.

Parents	Offspring
1011|1011	10111111
0010|1111	00101011

Figure 1-3. Crossover.

One crossover thus creates two new individuals, called offspring; one containing the beginning portion of the first individual followed by the ending portion of the second individual, and another containing the beginning portion of the second individual

followed by the ending portion of the first individual demonstrated in Figure 1-4. After some portion of the population is crossed-over, we have a new population of individuals,

Critter's String	Fitness	Selection Probability
X_1 = 10111111	7	7/21
X_2 = 10111010	5	5/21
X_3 = 00101011	4	4/21
X_4 = 00111011	5	5/21

Figure 1-4. Population after crossover (X_1, X_2, and X_3, X_4)

each of which is either identical to an individual in the prior population or is the product of genetic recombination through crossover. The significance of the crossover is explained by Holland "the purpose of crossing strings in the genetic algorithm is to test new parts of target regions rather than testing the same string over and over again in successive generations." [1].

Before evaluating the new population, one final genetic operator is applied: mutation. Mutation involves the flipping (switching 0 to 1 and vice versa) of alleles. A probability p_m (which is usually rather low) is defined as the chance of any given allele being flipped. In our One Max example, let us set $p_m = 0.05$. Since there are four individuals, each with eight loci, we would expect $(4)(8)(p_m) = 1.6$ mutations to occur. We will say that two mutations occur, in locus two of X_1 and in locus seven of X_4. We thus have the resulting critters 11111111 and 00111001.

After mutation, out new population is in its final state illustrated in Figure 1-5. The fitness values of the new individuals are evaluated by the objective function, and the new population is designated the current population, from which future generations will derive. As long as the completion criterion is not met, the three-step process of reproduction, crossover, and mutation is repeated. The completion criterion is generally either a perfect solution or a predetermined number of generations. In our rather simplistic One Max example, a fortunate sequence of events yielded a perfect solution after only one generation. In a more realistic application, it would not be unusual for the algorithm to continue for two hundred generations or more. When the algorithm does conclude, it gives as its solution to the optimization problem the individual in the final population with the highest fitness rating.

Critter's String	Fitness
$X_1 = 11111111$	8
$X_2 = 10111010$	5
$X_3 = 00101011$	4
$X_4 = 00111001$	4

Figure 1-5. Final population after mutation.

The repeated application of these three operators, each inspired by some aspect of natural selection, can thus solve some optimization problems. The reasons for the effectiveness of these operators are fairly clear. Building blocks (contiguous sequences of alleles)

which are beneficial to an individual are recombined through crossover with other individuals' building blocks from different loci within the chromosomal string. Since more fit strings are selected more frequently for reproduction and crossover, the more fit building blocks will join to form better and better solutions. Mutation serves to reintroduce diversity into the population, thus insuring that no alleles are lost. In our One Max example, for instance, none of the original individuals contained a 1 at the second locus. Mutation of the second allele in some individual was therefore necessary before the perfect 11111111 chromosome could be produced [9-11].

While the GA has achieved some definite success, it has its limitations. Things are not so simple that in order to solve any optimization problem, all we need to do is represent and evaluate individual solutions. The first difficulty is that the computation of objective fitness is non-trivial. It needs to be something a computer can do relatively quickly, since thousands, even millions, of individuals will need to be evaluated in the process of evolving better and better critters.

There are also many complications involved in the representation of individuals. If a representation is not chosen carefully, there could easily fail to be a one-to-one correspondence between genotypes and phenotypes; i.e., between representation of solutions to the problem and actual solutions. Careless representation schemes can also nullify the effectiveness of the crossover operator; it is possible that crossover would no longer serve to recombine useful parts of pairs of individuals, and even that crossover could create a chromosome which does not represent a legitimate solution [12]. In the processing of generating new generation from old generation using the three operators of GA, we have a big chance of losing the best point (or chromosome). In order to

circumvent this problem, a method called elitism is adapted. Elitism first copies the best point to new population. The rest is done in classical way. Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution.

1.2 Principle of Chemotaxis

In the fall of 1971 Max Delbrück [13] gave a lecture at Berkeley that described the peculiar behavior of chemotactic bacteria. They dash ahead in a more or less straight line, then tumble all over themselves, then dash off in a seemingly random direction, tumble again, etc. The dashes on which the concentration increases tend to be longer than dashes in the "wrong" direction. Intuitively it is clear that the net effect is that each bacterium migrates towards greater concentrations of the attractant. Professor Hans Bremermann at Berkeley realized at once that the behavior reported by Delbrück is equivalent to the steps of an optimization algorithm that he had reported earlier [3]. In both cases a maximum is sought, i.e. the maximum of a chemical concentration and the maximum of a function, respectively. The details of the optimization algorithm, however, vary greatly and there is very extensive literature. Many algorithms compute the gradient of a function and then proceed in the direction of the gradient (steepest descent). Some algorithms take successive directions to be orthogonal (conjugate gradient methods) to avoid certain difficulties than arise in some cases when the algorithms always follow the gradient. All these methods converge to local maxima or minima [14].

The chemotaxis algorithm performs a random-based search to find a set of parameter values which gives an objective function its lowest error. Two sets of parameters are

used, one set containing the values which have given the lowest error so far and a second set containing updated parameter values. The updated value set is produced by multiplying a random vector by a number, called the step size, and adding it to the lowest error set. The errors produced by the two parameter sets are compared after each update. If the updated set produces the lowest error, then it replaces the previous lowest error set. The same random vector is then used for successive updates, until it produces an updated set with a larger error. When an updated set produces a larger error, it is discarded and a new random vector is generated. Chemotaxis can alter the rate of convergence to the lowest error by altering the step size. If a particular random vector has produced a set of parameters with a lower error a number of times, then the step size is increased, since the direction on the error surface produced by the random vector is towards an area of low error. Hence convergence speed is increased. If a number of different random vectors have failed to produce a parameter set with a lower error, then the step size is reduced. It is assumed that the lowest error lies within the region described by a circle about the current lowest error, with radius given by the step size. Hence, by reducing the step size, chemotaxis can converge approximately to the lowest error without overshooting it [15, 16]. The application of using chemotaxis could be further found in references 17 – 19.

The chemotaxis algorithm can be described as working in the following steps when it is used to train a neural network:

Step 1. Initialize weights and biases of the network with small random values.

Step 2. Present the inputs to the network, and propagate data forward to obtain the predicted output.

Step 3. Determine the objective function over the whole data set.

Step 4. Generate a random vector for changes of weights and biases.

Step 5. Increment the weights and biases with changes.

Step 6. Calculate the new objective function.

Step 7. If the latter objective function is an improvement on the former then retain the modified weights and biases, and go to Step 5. If there has been no improvement then go to Step 4.

1.3 Principle of Simulated Annealing

Even though the simulating annealing technique is not used in this paper, there exist some similarities between Chemotaxis and simulated annealing. Publications based on the simulated annealing or its hybridized with genetic algorithms could be found in references [20 - 23]. Now I briefly introduce simulated annealing algorithm here, for further reference, see references [24 - 26].

Annealing is a term from metallurgy. When the atoms in a piece of metal are aligned randomly, the metal is brittle and fractures easily. In the process of annealing, the metal is heated to a high temperature, causing the atoms to shake violently. If it were cooled suddenly, the microstructure would be locked into a random unstable state. Instead, it is cooled very slowly. As the temperature drops, the atoms tend to fall into patterns that are relatively stable for that temperature. Providing that the temperature drop is slow enough, the metal will eventually stabilize into an orderly structure.

Simulated annealing can be performed in optimization by randomly perturbing the independent variables (weights in the case of neural network) and keeping track of the best (lowest error) function value for each randomized set of variables. A relatively high

standard deviation for the random number generator is used at first. After many tries, the set that produced the best function value is designed to be the center about which perturbation will take place for the next temperature. The temperature (standard deviation of the random number generator) is then reduced, and new tries done. The algorithm is summarized as following [27]:

1) Randomly generate an initial point S with a set of parameters.
2) Set the initial S to be the best-so-far point S*, thus S* = S.
3) Compute the cost of S, say C(S).
4) Compute the initial temperature T_0.
5) Set the temperature $T = T_0$.
6) While stop criterion is not satisfied do:

(a) Repeat M times:

(i) Select a random neighbor S' to the current S.

(ii) Set $\Delta C = C(S') - C(S)$.

(iii) If(ΔC) ≤ 0 (downhill move):

- Set S = S'.
- If(C(S) < C(S*)) then set S* = S.

(iv) If ($\Delta C > 0$) (uphill move):

- Choose a random number r uniformly from [0,1].
- If $r < e^{-\Delta C/T}$, then set S = S'.

(b) Reduce temperature T.

How do we progress from the starting temperature to the stopping temperature? One method is by multiplying by a constant factor each time. This factor is computed as

STATE UNIVERSITY

$$c = e^{\ln(stop/start)/(n-1)}$$

where start and stop are starting and stopping temperatures, and n is the number of temperatures.

1.4 Artificial Neural Networks

Since the artificial neural network is used to test the proposed algorithm, it necessitates the brief introduction of neural networks before we discuss any detail of the proposed algorithm.

An artificial neural network (ANN) is an information-processing system that is based on generalization of human cognition or neural biology. Fausett[28] gives these assumptions in common between the two:

- Information processing occurs at many simple elements called neurons.
- Signals are passed between neurons over connection links.
- Each connection link has an associated weight, which, in a typical neural net, multiplies the signal transmitted.
- Each neuron applies an activation function to its net input to determine its output signal.

A neural network is characterized by its particular:

- Architecture; its pattern of connections between the neurons.
- Learning Algorithm; its method of determining the weights on the connection.
- Activation function; which determines its output.

The processing elements considered in the definition of ANN are usually organized in a sequence of layers, with full connections between layers. Typically, there are three or

more layers: an input layer where data are presented to the network through an input buffer, an output layer with a buffer that holds the output response to a given input, and one or more intermediate or hidden layers as shown in Figure 6 [29].

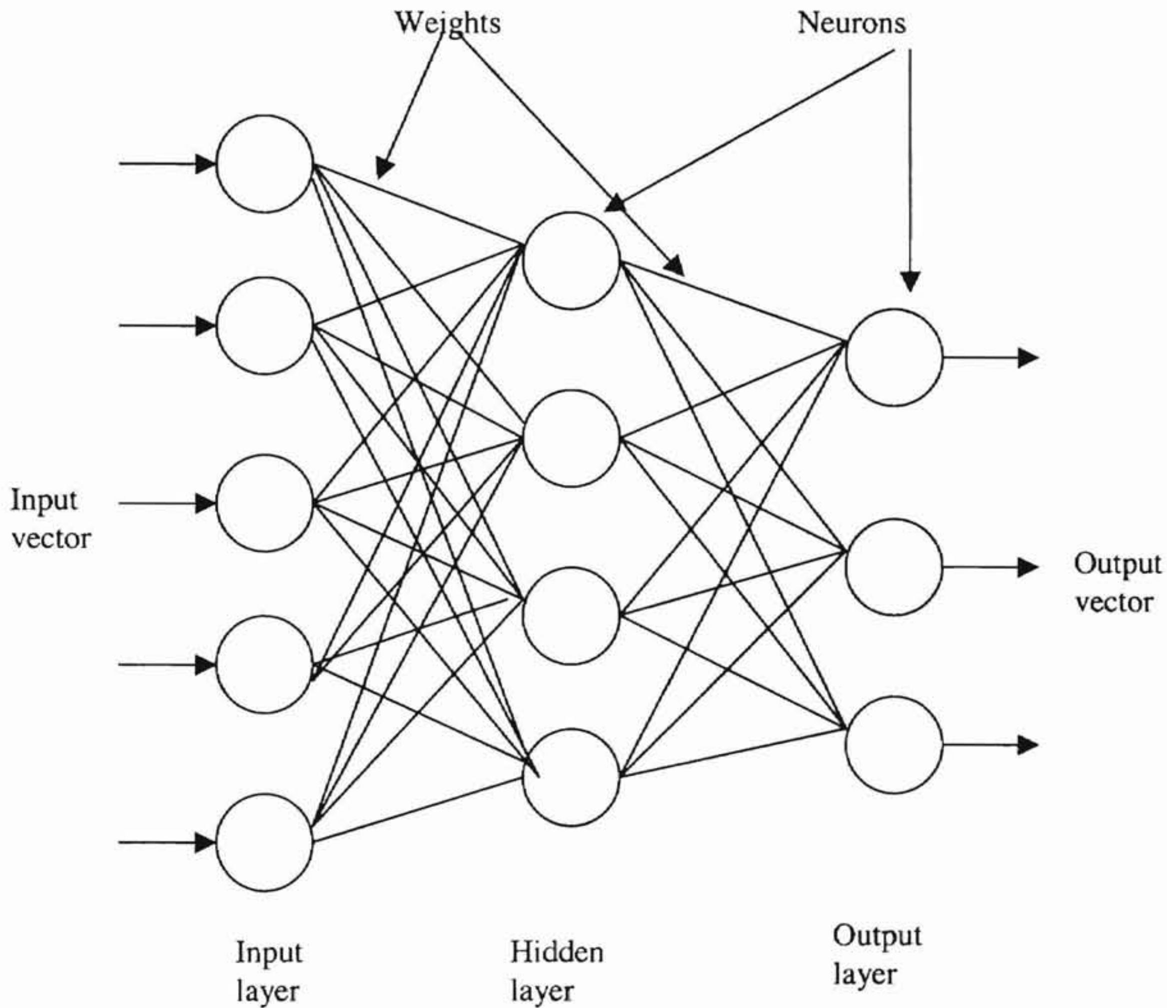


Figure 1- 6. Artificial Neural Network.

The operation of an ANN involves two processes: learning and generalization. Learning is the process of adapting the connection weights in response to external stimuli at the

input buffer. The network "learns" in accordance with a learning rule governing the adjustment of connection weights in response to learning examples applied at the input and output buffers. Generalization is the process of accepting an input and producing a response determined by the geometry and synaptic weights of the network.

Each hidden neuron provides an additive contribution to the input of the neuron with which it is connected. The total input to a neuron is simply the weighted sum of the separate outputs from each of the connected neurons plus a bias or offset term θ_i :

$$i_i(t) = \sum_j w_{ij}(t)a_j(t) + \theta_i(t) \qquad (1.1)$$

where a_j is current state of neuron j and each w_{ij} is the weight of the connection between neurons i and j. A positive weight is considered as an excitation and a negative weight an inhibition.

It is necessary to have a rule which gives the effect of the total input on the activation of the neuron. This rule is a function F_i which takes the total input $i_i(t)$ and current activation $a_i(t)$ and produces a new value of the activation of the neuron i:

$$a_i(t+1) = F_i(a_i(t), i_i(t)) \qquad (1.2)$$

Often, the activation function is a nondecreasing function of the total input of the neuron:

$$a_i(t+1) = f_i(i_i(t)) = F_i(\sum w_{ij}(t)a_j(t) + \theta_i(t)) \qquad (1.3)$$

although activation functions are not restricted to nondecreasing function. Generally, some sort of threshold function is used: a hard limiting threshold function, or a linear or semi-linear function, or a smoothly limiting threshold. A sigmoid (S-shaped) function for this smoothly limiting function is often used, for example:

$$a_i = F(i_i) = 1/(1+e^{-i}{}_i) \qquad (1.4)$$

In all networks the output of a neuron is considered to be identical to its activation level.

Network topologies are divided into the following groups [30]:

- Feed-forward networks, where the data flow from input to output neurons is strictly feed-forward. The data processing can extend over multiple (layers of) neurons, but no feedback connections are present, that is, connections extending from outputs of neurons to inputs of neurons in the same layer or previous layers.
- Recurrent networks, which do not contain feed back connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the neurons undergo a relaxation process such that the network will evolve to a stable state in which these activations do not change anymore. In other applications, the change of the activation values of the output neurons are significant, with the dynamic behavior constituting the output of the network.

The learning algorithm plays an important role in any NN. This is the process of modifying the weights and biases to the neurons. Typically, we do not know what the output space will look like in advance. The NN must be trained to classify certain data patterns to certain outputs. In the process of training, the weights on the neural connections change, and thus the output decision boundaries change during training. The learning situations of NNs can be categorized in these two paradigms:

- Fixed weights, so that no learning occurs.
- Supervised learning or associative learning, where each input vector is associated with a target output vector.
- Unsupervised learning or self-organization, where no target outputs are specified.

Typically, training is continued until a preset condition is met. This may be, for example, minimization of a defined error function. One full pass through the training set is termed an epoch. Sometimes training is performed until a set number of epochs have been completed.

Training is performed on an NN so that it will correctly identify input patterns. By training an NN we separate the output space into regions. Of course, the output space will not only separate (classify) the input data patterns, but it will also separate data patterns which it has not seen before. The ability of an NN to classify input data patterns correctly that it has not seen before (has not been trained with) is termed generalization. A net that has been overtrained will usually have poor generalization, since the output space will follow the training data too closely.

The input patterns must be chosen so that they display the particular features one would like the net to learn. They are prepared in an N-dimensional array which is fed into the N input neurons of the input layer. It is important to limit the number of variable, used in the input patterns, since the actual training of a neural network is very time consuming. Finally one should make sure that the input variables are normalized, to avoid saturating the activation functions [31, 32].

CHAPTER II METHODOLOGY

During the past decades, the role of optimization has steadily increased in such diverse areas as, for example, electrical engineering, operation research, computer science, and communications [33]. Optimization problems are very important to production and our daily life. In practice optimization problems become more and more complex. For example, many large scale combinatorial optimization problems can only be solved approximately on present-day computers, which is closely related to the fact that many of these problems have been proven to be NP-hard [34]. Deterministic polynomial time algorithms for their solution are unlikely to exist. The quality of the final solution is not improved computation time. In some continuous optimization problems, the search for an optimum of a function of continuous variables is difficult if there are peaks and valleys, ruts and ridges. In these cases, traditional optimization methods are not effective. They either become trapped in local minima or need much more search time. In recent years, many researchers have tried to find some new ways to solve these difficult problems. Stochastic approaches have attracted much attention [35].

Genetic algorithms (GA) and the chemostaxis algorithm (CA) are all stochastic algorithms. Stochastic algorithms have some good characteristics. Many results have been presented [36]. Although stochastic algorithms have been successfully used in some difficult cases, there are still some problems. Based on the analysis and applications of GA and CA, we propose a new stochastic algorithm called GADCA (GA:

genetic algorithm with diversity guidance, CA: chemotaxis algorithm) which integrates the advantages of GA and CA. It has high search speed and precision.

2.1 Comparison of GA and CA

A. The main features of a GA are summarized as:

1) GAs work from a population instead of a single state. The population evolves by the use of operators such as crossover, mutation and so on.
2) Good individuals with a higher fitness value always have a better chance of producing offspring. In contrast, bad individuals with low fitness value still have a chance to reproduce.
3) The mutation operator can introduce some new information into a generation. The probability of escaping from local minima of a GA is higher than that of a CA.

A GA is effective for many optimization problems, but it still has difficulties such as premature convergence and evolving too slowly. Many advanced genetic algorithms have been presented in the literature, but their complexity is also increased over traditional genetic algorithms.

B. The key features of the CA are shown as follows:

1) The CA is very simple and easy to use.
2) The CA only accepts good states which have lower search cost. It can converge rapidly, but it has more difficulty escaping from local minima than do GAs.
3) The CA uses Gaussianly (normally) distributed variables to generate new states, so it is not suitable for optimizing discrete problems.

After analyzing the search process of stochastic algorithms we find that there are two kinds of search in a stochastic optimization method. They are "directed search" and "blind search". For example, in the search process some algorithms mainly accept new states corresponding to a decrease in cost function. This kind of search is directed. Sometimes the search process accepts bad states randomly; this kind of search process is blind. "Blind search" enables the search process to escape from local minima. Therefore, if these two kinds of search cooperate properly, the optimization algorithm will have good properties of inheriting the advantages of genetic algorithm and chemotaxis method, respectively. The combination can be made by generating some of the new points by a genetic algorithm with diversity guidance (to be discussed in chapter 3) and some by the chemotaxis method. A point is a complete neural network structure consisting of weights. The proportion of points is determined by following two equations as the global optimum is approached:

$$P_c = k/k_m \qquad (2.1)$$

$$P_g = 1 - P_c \qquad (2.2)$$

Where P_g is the proportion of the points generated by the genetic algorithm with diversity guidance and P_c the proportion by the chemotaxis method. k is the generation sequential number, and k_m is the maximum number of generations expected. From the beginning of the search, a very low proportion of points are allowed to be generated by the chemotaxis method, because their parents are far from the global optimum. As the search progresses, the points gradually approach the global optimum and then a high proportion of points generated by the chemotaxis method are needed to speed up convergence. Based on the above analysis, we present a novel algorithm GADCA.

2.2 The Proposed Algorithm GADCA

A. The outline of GADCA

Step 1. Randomly initialize n points from the search space with equal probability.

Step 2. Calculate the objective function values of the n points.

Step 3. Sort the n points in the order of increasing objective function values, so that the first point represents the best and the last point represents the worst.

Step 4. Each of the points is assigned a probability p_i, i=1, 2, 3, ..., n, giving a higher probability to the points with lower function values and lower probabilities to those with higher function values.

Step 5. Randomly select two different points from n points according to the probability p_i.

Step 6. For each of the genes or weights, randomly select one value from the corresponding two selected points to construct a new point.

Step 7. For each of the genes of the newly created point, generate a random number r, if $p_n > r$; then replace the value of that gene by another random number.

Step 8. Repeat k times Step 5-7 so that k new points are generated (steps 5 – 7 are genetic algorithm steps).

Step 9. Randomly generate a point, multiply each gene of the point by a number called step size (for example 0.01). Add each gene of the point to the corresponding gene of the best-so-far point resulting in an updated point.

Step 10. If the updated point is better than the best-so-far point, keep the point randomly generated and multiply each gene by the step size until the updated point is worse than the best-so-far point. Add the updated point just before it fails into the new population.

Step 11. Repeat step 9 – 10 n-k time's to obtain the size of the new generation is the same size as that of its parents n (step 9 –10 are chemotaxis algorithm).

Step 12. Calculate the objective function values for the newly created points.

Step 13. Sort the newly created points into ascending order.

Step 14. If the best point of the new generation 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 resort them. This step is to ensure that the current best-so-far point in the community is always retained.

Step 15. Start from the next-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 further away from the best-so-far point, then keep the new one; then compare the rest until they are all finished; go to Step 18; otherwise, go to Step 16.

Step 16. If the distance of the old point is further away from the best-so-far point and has better fitness, then keep the old one and reject the new one and go to Step 15 to screen others; otherwise, go to Step 17.

Step 17. 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_o is greater than the distance of the old one d_o times the objective function value of the new one f_n (i.e., $d_n f_o > d_o f_n$), then select the new one and go to Step 15. 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 (introduction of diversity).

Step 18. Use the new population as a new generation, repeat Step 3 to Step 18 until either a predetermined iterative number or an acceptable objective function value is reached.

B. The Features of GADCA

1) GADCA works from a population, which takes the advantage from GA. Search from many states simultaneously is more efficient than search from a single point. It is easier to find the global optimum.
2) GADCA should converge fast in terms of the combination of the CA. At very beginning of the training process, the GA plays dominant role. When the search is approaching the global minima, the CA starts functioning. In the CA portion, only a decreased objective function value is accepted, which strengthens the local search around the best state of the populations. It is helpful to find the global optimum.
3) GADCA is better than pure GA. It only needs a small population size due to the introduction of the diversity shown from Step 14 to Step 17. This introduction of the diversity dramatically reduces the memory space requirement for the storage of the population.

2.3 Case Studies and the Selection of Main Parameters

In the investigation of GADCA, two cases in a variety of areas are used to verify the reasonability, correctness and effectiveness of GADCA. In addition, the selection of main parameters such as mutation probability p_m, step size s and population size m will be studied. Their effect upon the performance of the GADCA will be explored. The multilayer neural networking architecture is utilized to investigate the GADCA. In all cases, the objective function $(\Sigma(y_i - x_i)^2)^{1/2}$/(number of data points – number of parameters) is maintained to evaluate the performance of the network, where y is the computed output, x the target output, and i = 0, 1, 2, 3 ,..., n. The program is written in C++.

CHAPTER III RESULTS AND DISCUSSIONS

The procedure described in chapter 2 reflects natural genetics in some respects. For any animal species, the DNA chain of an individual is a mixture of the DNA chain of its parents. Furthermore, fit parents are likely to produce fit offspring, and better performing individuals have a better chance of surviving and producing more offspring than worse ones. In any case, the individual with the best adaptation remains in the population at the expense of the weaker individual, until an individual with superior adaptation replaces it. The combination of genetic algorithm and chemotaxis searching takes care of both the genetic algorithm, which makes the searching globally optimum, and of the chemotaxis method, which converges quickly as it approaches the optimum. The following case studies will demonstrate the ideas and features of the proposed algorithm.

3.1 Case 1. The first case is the application of the proposed algorithm to a chemical engineering problem. The training data is listed in Table 3-1 [36]. The temperatures and pressures are the input data for the neurons in the input layer. The logarithm of the viscosity, measured at different temperature and pressure, is the target output for the comparison of the computed output from the neuron in the output layer. According to the analysis of the data, the neural network architecture consists of three layers with two neurons in the input layer, three neurons in the hidden layer and one neuron in the output layer. There exist nine weights and four biases in this structure. Unfortunately, there is no report of trying to fit these data using a non-neural model. Otherwise, it would have

given a good reference for my investigation. The purpose of my investigation is to compare the results of using simplest genetic algorithm, chemotaxis algorithm, and their scientific combinations, not to find the best search method. There is no doubt that there exit a lot of algorithms, such as damped Newton method [37] and quasi-Newton method [38], maybe resulting in better results. The number of neurons selected for in the hidden layer may be optional. However, the number of total weights in the neural network must be not larger than the number of data items in the training data set. Otherwise, an overfit condition will occur. When the neural network is constructed of many hidden layers, it creates not only a complicated network structure, but also slows the process of training the neural network without enhancing the performance. The objective function is $(\Sigma(y_i - x_i)^2)^{1/2}$/(number of data points – number of parameters), where y is the computed output, x is the experimental output which is the value in the last column in the Table 1-1. The goal of training the neural network is to minimize the objective function value using the proposed algorithm through adjusting the weights of the network.

Table 3-1. Training data for the lubricant viscosity at different temperature and pressure [37].

Sample number	Temperature (^{0}C)	Pressure (atm)	ln(viscosity) (experimental)
1	0.0	1.0	5.106
2	0.0	740.8	6.387
3	0.0	1407.5	7.385

Table 3-1 continued.

4	0.0	363.2	5.791
5	0.0	1.0	5.107
6	0.0	805.5	6.361
7	0.0	3907.5	11.927
8	0.0	4125.6	12.426
9	0.0	2572.0	9.156
10	25.0	1.0	4.542
11	25.0	805.0	5.825
12	25.0	1505.9	6.705
13	25.0	2340.0	7.716
14	25.0	422.9	5.298
15	25.0	5064.3	11.984
16	25.0	5280.9	12.444
17	25.0	3647.3	9.523
18	25.0	2813.9	8.345
19	37.8	516.8	5.173
20	37.8	1738.0	6.650
21	37.8	1008.7	5.807
22	37.8	2749.2	7.741
23	37.8	1375.8	6.232
24	37.8	191.1	4.661
25	37.8	1.0	4.298

Table 3-1 continued.

26	37.8	4849.8	10.811
27	37.8	5605.8	11.822
28	37.8	6273.9	13.068
29	37.8	3636.7	8.804
30	37.8	1949.0	6.855
31	37.8	1298.5	6.119
32	98.9	1.0	3.381
33	98.9	686.0	4.458
34	98.9	1423.6	5.207
35	98.9	2791.4	6.291
36	98.9	4213.4	7.327
37	98.9	2103.7	5.770
38	98.9	402.2	4.088
39	98.9	1.0	3.374
40	98.9	2219.7	5.839
41	98.9	6344.2	8.914
42	98.9	7469.4	9.983
43	98.9	5640.9	8.323
44	98.9	4107.9	7.132

3.1.1 Comparison of Using Diversity and without Diversity

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

To measure diversity, the Euclidean distance between two points,

$$d = (\Sigma(x_i - y_i)^2)^{1/2}/(\text{number of data points} - \text{number of parameters})$$

is used, where x_i and y_i are the i-th values of the points x and y, respectively. Obviously, the larger the value of d, the greater the distance between the two points. For example, $d = 0$ implies the two points are identical, that is, there is no difference between them. Thus, to keep one of them in the population 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. Reference 27 illustrates the improvement of genetic algorithm performance in terms of the introduction of diversity based upon consideration of distance between two points.

OKLAHOMA STATE UNIVERSITY

In order to compare the difference between the pure genetic algorithm and the genetic algorithm with the introduction of diversity, the procedure for training the neural network is carried out using the two algorithms, respectively. When the mutation probability p_m = 0.01, different population sizes are used and the ten-run-average best-so-far objective function value is calculated for various numbers of objective function evaluations.

Tables 3-2 - 4 depict two attractive advantages of using diversity guidance against without using diversity guidance for genetic algorithm. First, performance is different when genetic diversity guidance is introduced. The efficiency of the genetic algorithm is remarkably improved. Figures 3-1 to 3 show that objective function evaluation with genetic diversity guidance produces a much better result than objective function evaluation without genetic diversity guidance. In Figures 3-1 to 3, the vertical axis represents the objective function values or errors, horizontal axis represents population size. When the genetic algorithm with the diversity guidance is introduced, the objective function value or error decreases dramatically with compared to without the introduction of diversity guidance in all cases of different epochs.

Second, when genetic diversity guidance is used, the genetic algorithm prefers a smaller population size, rather than larger size. When the population size is large enough, the

Table 3-2. Comparison of the impact of using genetic diversity guidance for the genetic algorithm. Mutation probability p_m = 0.01, population size m, Epoch = 100.

m	a*	b*
2	10.535	0.417
3	6.948	0.368

OKLAHOMA STATE UNIVERSITY

Table 3-2 continued.

4	3.147	0.245
5	2.360	0.239
6	2.344	0.235
10	1.148	0.190
20	0.428	0.098
50	0.340	0.107
80	0.293	0.162
100	0.280	0.197

a* : not using genetic diversity guidance; b* : using genetic diversity guidance.

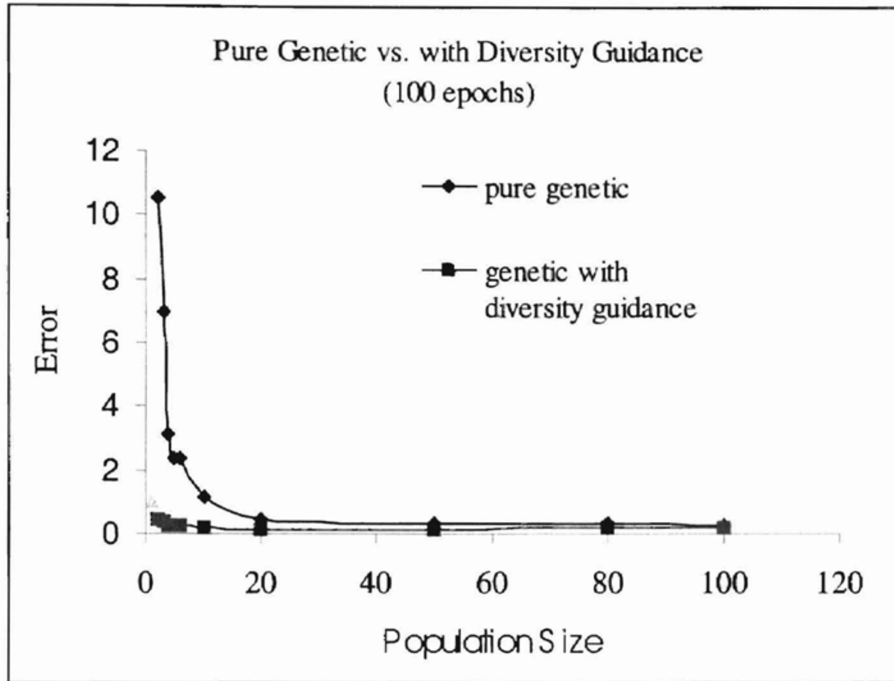


Figure 3-1. Effect of the variation of population size on the objective function value (error) for pure genetic algorithm and genetic algorithm with diversity guidance, respectively, when the epoch = 100.

Table 3-3. Comparison of the impact of using genetic diversity guidance for the genetic algorithm. Mutation probability $p_m = 0.01$, population size m, Epoch = 1000.

m	a*	b*
2	7.454	0.192
3	4.531	0.136
4	4.180	0.091
5	2.571	0.070
6	2.326	0.088

Table 3-3 continued.

10	0.959	0.095
20	0.570	0.112
50	0.280	0.131
80	0.278	0.141
100	0.263	0.150

a* : not using genetic diversity guidance; b* : using genetic diversity guidance.

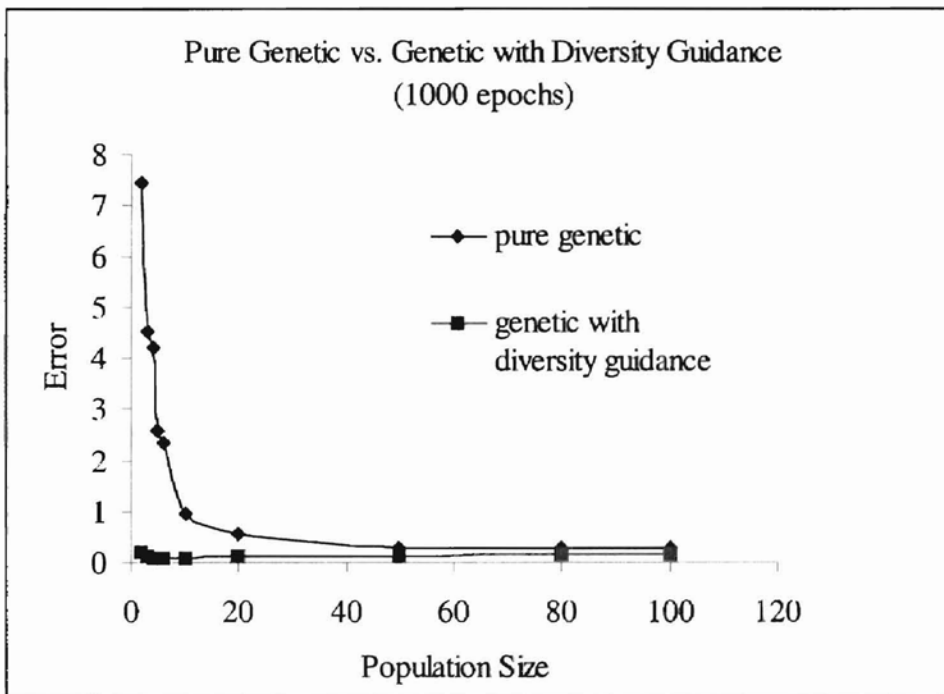


Figure 3-2. Effect of the variation of population size on the objective function value (error) for pure genetic algorithm and genetic algorithm with diversity guidance, respectively, when the epoch = 1000.

Table 3-4. Comparison of the impact of using genetic diversity guidance for the genetic algorithm. Mutation probability $p_m = 0.01$, population size m, Epoch = 2000.

m	a*	b*
2	19.042	0.316
3	5.481	0.086
4	4.927	0.085
5	2.270	0.054
6	1.970	0.068
10	0.878	0.082
20	0.452	0.095
50	0.290	0.128
80	0.274	0.140
100	0.268	0.143

a* : not using genetic diversity guidance; b* : using genetic diversity guidance.

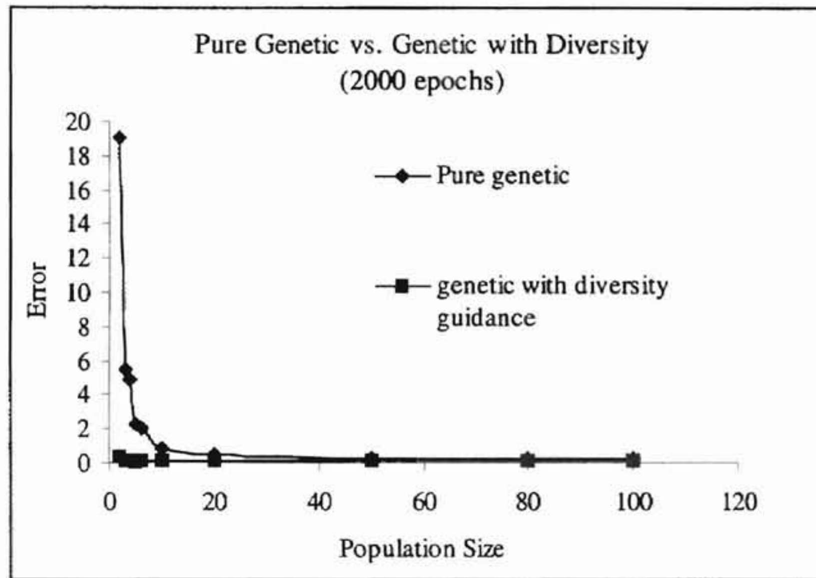


Figure 3-3. Effect of the variation of population size on the objective function value (error) for pure genetic algorithm and genetic algorithm with diversity guidance, respectively, when the epoch = 2000.

efficacy of the genetic diversity guidance is damped because a large population size can contain almost every possible character. When epoch is given, the error increases beyond a certain population size for genetic algorithm with diversity guidance. As we notice, each point consists of the random generated parameters. The larger the population size, the more the parameter variation is. The possibility of introducing larger parameters in generating a new point also increases, as a result, the error enhances. That may be the reason why traditional genetic algorithms need a very large population 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

population and slow down the approach to the global minimum. Therefore, no matter how large the population size is, the introduction of diversity guidance can improve the efficiency of the genetic algorithm.

3.1.2 Comparison of Genetic Algorithm with Diversity (GAD) and Genetic Algorithm with Diversity Combined with Chemotaxis (GADCA)

For the same problem, set $p_m = 0.01$, and use equations 2.1 and 2.2 to control the proportion of new points generated by the genetic algorithm and the chemotaxis method. Table 3-5 shows clearly that a combination with the chemotaxis method can further improve the efficiency of the pure genetic algorithm, especially when a more accurate result is required. When the generation increases, the objective function value decreases in both cases of GAD and GADCA. However, the objective function value decreases much faster for GADCA than that for GAD. This is because the genetic algorithm only drives the points in the vicinity of the global minimum. The rest of the work may be left for the chemotaxis method to finish.

Table 3-5. Comparison of the genetic algorithm with diversity guidance (GAD) and the genetic algorithm with diversity guidance combined with chemotaxis (GADC), case 1, $p_m = 0.01$, $p= 5$, $s = 0.0001$.

	Ten-run-average, best-so-far objective value	
Generations	GAD	GADCA
50	0.392	0.096
100	0.305	0.056

Table 3-5 continued.

200	0.221	0.048
400	0.157	0.031
600	0.093	0.029
1000	0.086	0.027

3.1.3 Comparison of Genetic Algorithm with Diversity Guidance (GAD), Chemotaxis Algorithm (CA) and Genetic Algorithm with Diversity Guidance Combined with Chemotaxis (GADCA)

For comparison, the GAD, CA and GADCA performances are conducted under different generations when $p_m = 0.01$. Table 3-6 shows that GADCA can gradually reach the global minimum. As the generation grows, the probability of reaching the global minimum is increased. Even though GAD converges gradually, it shrinks slower than GADCA, which includes the chemotaxis algorithm. In contrast, chemotaxis working on a single point converges as the generation increases; the speed of convergence is much slower than GAD and GADCA. It could be rationalized that it lacks a global minimum since it only works on a single point, the global minimum is not guaranteed. The possibility of becoming trapped in a local minimum cannot always be avoided. Meanwhile, for each generation only one step closer to the minimum could be obtained resulting in a slower convergence since chemotaxis only works on single point for each generation. On the other hand, both GAD and GADCA inherit the advantage of the natural adaptation character of which the smaller the objective function value for a point

is, the higher the probability for the point to survive. For each generation, a better-fit group of offspring is obtained for the whole population resulting in a faster step to the minimum. The introduction of diversity in both algorithms produces the global minimum.

Table 3-6. Comparison of the genetic algorithm with diversity guidance (GAD), chemotaxis algorithm (CA) and the genetic algorithm with diversity guidance combined with chemotaxis (GADCA), case 1, $p_m = 0.01$, $p = 5$, $s = 0.0001$.

	Ten-run-average, best-so-far objective value		
Generations	GAD	CA	GADCA
50	0.392	0.270	0.096
100	0.365	0.249	0.056
200	0.221	0.246	0.048
400	0.157	0.228	0.031
600	0.093	0.197	0.029
1000	0.086	0.156	0.027

3.1.4 Sensitivity of the Parameters of GADCA

GADCA is quite a simple algorithm, and easy to perform; however, there are a few parameters required. GADCA has three parameters of its own, the population size m, mutation probability p_m and the step size s. From the results obtained by varying the

population size on the performance of the pure genetic algorithm and the genetic algorithm with diversity guidance, we can determine that the population size should be smaller when the GADCA is conducted. The mutation probability, which controls the change of the genes after the new point is selected, should be appropriately selected, otherwise, the performance of the proposed algorithm will be deteriorated. The step size of chemotaxis used to modify the best-so-far point is conducted to obtain one-step close to a better point than the best-so-far point. However, there is no step size introduced in a genetic algorithm. The step size of chemotaxis algorithm will give rise to a very long training time if it is very small. If the step size is very large it will cause convergence failure. It is possible that the selection of parameters for a given algorithm may be problem related. However, there should be some general guidelines.

To investigate the sensitivity of the parameters for the proposed algorithm, the three parameters are used and the ten-run-average, best-so-far objective function values are calculated. The results for 100 and 1000 epochs evaluations in Table 3-7 illustrates following points.

1. The proposed algorithm generally is not very sensitive to the parameters. Therefore it is robust, and may be applied successfully in many conditions.
2. When the mutation rate is 0.05 and other parameters keep constant, the proposed algorithm yields the objective function value. It is evident that the proposed algorithm performs better with a smaller mutation probability value. However, when the probability is less than 0.01, the generations end prematurely due to lack of great diversity of points.

3. It is observed that the proposed algorithm gives the best objective function value even though the objective function values are not very significant when the population size is varied.
4. The algorithm is relatively more sensitive to step size than the other two parameters m and p_m. If s is extremely small, updating the best-so-far point will take many steps to finish until it fails. It is detrimental to the efficiency of the algorithm. On the other hand, if s is large, finding a better point to update the best-so-far point will be impossible. Thus, the algorithm will be in efficient in terms of finding a better point to replace the best-so-far point. In this case, when s = 0.0001, the algorithm presents the best performance.

Table 3-7. Influence of varying the parameters on the performance of GADCA, case 1, ten-run-average, best-so-far objective function values.

Effect of varying mutation probability p_m for m = 5, s = 0.0001 on performance						
P_m	0.01	0.05	0.10	0.20	0.30	0.50
100 epochs	0.056	0.049	0.064	0.073	0.074	0.076
1000 epochs	0.027	0.013	0.028	0.035	0.038	0.038
Effect of varying step size s for m = 5, p_m = 0.05 on the performance						
S	1.0	0.1	0.01	0.001	0.0005	0.0001
100 epochs	0.243	0.201	0.119	0.068	0.049	0.026
1000 epochs	0.096	0.060	0.039	0.027	0.022	0.021

Table 3-7 continued.

Effect of varying population size m for pm = 0.05, s = 0.0001 on the performance						
m	3	4	5	10	20	100
100 epochs	0.078	0.050	0.041	0.041	0.060	0.062
1000 epochs	0.045	0.023	0.011	0.017	0.018	0.020

3.1.5 The effective of varying the Number of Neurons in Hidden-layer on the Performance of GADCA

Further experiment is carried out when the number of neurons in the hidden-layer is varied. The neural network structures are 3:2:1, 2:2:1 and 2:1:1 (number of neurons in the input layer: that in the hidden-layer: that in output-layer). The experimental result is shown in Table 3-8. It is evident that the performance of neural network is better when the structures are 2:3:1 and 2:2:1 than that of 2:1:1. However, the standard deviation increases somewhat when neural network structure is 2:1:1.

Table 3-8. The influence of varying the number of neurons in the hidden-layer to the performance of GADCA. Number of points = 5, step size = 0.0001, mutation probability = 0.05. A: neural network structure 2:3:1, B: neural network structure: 2:2:1 and C: neural network structure 2:1:1 (number of neurons in the input layer: that in the hidden-layer: that in output-layer).

	Standard Deviation		
Generation	A	B	C
50	0.096	0.112	0.16
100	0.056	0.054	0.084
200	0.048	0.039	0.053
400	0.031	0.029	0.058
600	0.029	0.026	0.061
1000	0.027	0.023	0.06

3.1.6 Training Result of Data in Table 3-1 Using GADCA

Based upon the above discussion of the effect of varying the parameters on the performance of the proposed algorithm GADCA, the following parameters may be suggested:

1. Population size m = 5,
2. Mutation probability $p_m = 0.05$,
3. Step size s = 0.0001.

The neural network architecture is still a three-layered structure, two neurons in the input layer, three neurons in the hidden layer and one neuron in the output layer. There exist six weights and four biases in the structure. The objective function is $(\Sigma(y_i - x_i)^2)^{1/2}$/(number of data points – number of parameters), where y is the computed output, x is the experimental output. When the above suggested parameters are accepted, the proposed algorithm yields objective function value (error) of 0.0065 after 6000 generations. The computed results are listed in Table 3-9. If more accuracy is required, it is capable of utilizing more generations.

Table 3-9. Training result for the lubricant viscosity at different temperature and pressure in Table 3-1.

Sample Number	Temperature (^{0}C)	Pressure (atm)	ln(viscosity) (experimental)	ln(viscosity) (computed)
1	0.0	1.0	5.106	5.113
2	0.0	740.8	6.387	6.365
3	0.0	1407.5	7.385	7.425
4	0.0	363.2	5.791	5.741
5	0.0	1.0	5.107	5.113
6	0.0	805.5	6.361	6.469
7	0.0	3907.5	11.927	11.892
8	0.0	4125.6	12.426	12.375
9	0.0	2572.0	9.156	9.323
10	25.0	1.0	4.542	4.603

Table 3-9 continued.

11	25.0	805.0	5.825	5.780
12	25.0	1505.9	6.705	6.715
13	25.0	2340.0	7.716	7.757
14	25.0	422.9	5.298	5.255
15	25.0	5064.3	11.984	11.945
16	25.0	5280.9	12.444	12.389
17	25.0	3647.3	9.523	9.514
18	25.0	2813.9	8.345	8.362
19	37.8	516.8	5.173	5.097
20	37.8	1738.0	6.650	6.610
21	37.8	1008.7	5.807	5.745
22	37.8	2749.2	7.741	7.737
23	37.8	1375.8	6.232	6.191
24	37.8	191.1	4.661	4.625
25	37.8	1.0	4.298	4.330
26	37.8	4849.8	10.811	10.481
27	37.8	5605.8	11.822	11.802
28	37.8	6273.9	13.068	13.183
29	37.8	3636.7	8.804	8.779
30	37.8	1949.0	6.855	6.847
31	37.8	1298.5	6.119	6.100
32	98.9	1.0	3.381	3.417

Table 3-9 continued.

33	98.9	686.0	4.458	4.395
34	98.9	1423.6	5.207	5.218
35	98.9	2791.4	6.291	6.344
36	98.9	4213.4	7.327	7.268
37	98.9	2103.7	5.770	5.827
38	98.9	402.2	4.088	4.019
39	98.9	1.0	3.374	3.417
40	98.9	2219.7	5.839	5.920
41	98.9	6344.2	8.914	8.845
42	98.9	7469.4	9.983	10.060
43	98.9	5640.9	8.323	8.250
44	98.9	4107.9	7.132	7.201

3.1.7 Testing the Neural Network Using GADCA

Normally, after a neural network has been trained, it is tested using another data set to examine whether the neural network has been trained reasonably. The neural network was tested using the test data set in Table 3-10. The test results are shown in Table 3-10. The objective function value (error) is 0.013.

Table 3-10. Test data for the lubricant viscosity at different temperature and pressure [37].

Sample Number	Temperature (^0C)	Pressure (atm)	ln(viscosity) (experimental)
1	0.0	1407.5	7.385
2	0.0	3907.5	11.927
3	25.0	1505.9	6.705
4	25.0	5280.9	12.44
5	37.8	1375.8	6.232
6	37.8	3636.7	8.804
7	98.9	2791.4	6.291
8	98.9	7469.4	9.983

Table 3-11. Testing result for the test data set in Table 3-10.

Sample number	Temperature (^0C)	Pressure (atm)	ln(viscosity) (experimental)	ln(viscosity) (computed)
1	0.0	1407.5	7.385	7.424
2	0.0	3907.5	11.927	11.892
3	25.0	1505.9	6.705	6.714
4	25.0	5280.9	12.44	12.388
5	37.8	1375.8	6.232	6.190
6	37.8	3636.7	8.804	8.778

Table 3-11 continued.

7	98.9	2791.4	6.291	6.343
8	98.9	7469.4	9.983	10.059

3.1.8 Generalization of the Neural Network Using GADCA

Keeping the same neural network architecture as that in training and testing procedure and the suggested parameters, the neural network is extended to arbitrary data set in Table 3-12. The generalization results are shown in Table 3-13.

Table 3-12. Generalization data set [37].

Sample number	Temperature (^{0}C)	Pressure (atm)	ln(viscosity) (experimental)
1	0.0	1868.1	7.973
2	0.0	3285.0	10.473
3	25.0	1168.4	6.226
4	25.0	2237.3	7.574
5	25.0	4216.9	10.354
6	37.8	2922.9	7.957
7	37.8	4044.6	10.511
8	98.9	3534.8	6.726
9	98.9	4937.7	7.768

Table 3-13. Generalization results for the data in Table 3-12.

Sample number	Temperature (^0C)	Pressure (atm)	ln(viscosity) (experimental)	ln(viscosity) (computed)
1	0.0	1868.1	7.973	8.156
2	0.0	3285.0	10.473	10.619
3	25.0	1168.4	6.226	6.282
4	25.0	2237.3	7.574	7.627
5	25.0	4216.9	10.354	10.402
6	37.8	2922.9	7.957	7.932
7	37.8	4044.6	10.511	9.304
8	98.9	3534.8	6.726	6.837
9	98.9	4937.7	7.768	7.741

3.1.9 Conclusions for GADCA

From the above case study, some general conclusions can be drawn:

1. The genetic algorithm with diversity guidance is efficient, because it only needs a relatively small size population. This guidance dramatically reduces the memory size for storing the large population compared to when the pure genetic algorithm is applied.
2. The combination of genetic algorithm with chemotaxis is reasonable and effective. The effectiveness of the combination of genetic algorithm with chemotaxis can be found by comparing its objective function value with those of genetic algorithm and

chemotaxis, respectively (Table 3-6). For instance, when the generation = 100, the objective function value for combination of genetic algorithm with chemotaxis is 0.328. However, the objective function values for genetic algorithm and chemotaxis are 1.781 and 1.450, respectively. The rationale of the combination of genetic algorithm with chemotaxis can be suggested by the applicability of this algorithm in case 1. The similarity between the experimental viscosity and computed viscosity from the neural network trained by the combination genetic algorithm with chemotaxis method shown in Table 3-12 demonstrates that the proposed algorithm is reasonable. For instance, when the experimental data for viscosity are 7.973, 10. 473, 6.226, 7.574, 10.354 and 7.957, the computed data corresponding to the experimental data are 8.156, 10.619, 6.282, 7.627, 10.402 and 7.932.

3. The GADCA is tunable. By controlling the composition of the points in the population, the ratio of the points generated by the genetic algorithm with diversity guidance to those generated by chemotaxis is tunable. It can easily avoid a local minimum, reach the global minimum efficiently, and converge quickly.
4. Carefully selecting the parameters for GADCA, especially the step size, can make the algorithm more efficient.

3.2 Case 2. Application of GADCA for Water Pressure at Different Temperatures

The genetic algorithm with diversity guidance combined with chemotaxis (GADCA) is extended to the determination of water pressure at different temperatures. For the training data set in Table 3-13[39], the neural network architecture is a three-layer structure, one neuron in the input layer, two neurons in the hidden layer and one neuron

in the output layer. There are four weights and three biases in this structure. The temperatures in Table 3-14 serve as the input data. The pressures in Table 3-14 serve as the target data which are used to compare the difference between the computed data and the experimental data. The objective function is $(\Sigma(y_I - x_I)^2)^{1/2}$/(number of data points – number of parameters), where y is the computed output, x is the experimental output.

Table 3-14. Vapor pressure of water [39].

Sample Number	Temperature (^{0}C)	Pressure (mmHg)
1	10	9.2
2	11	9.8
3	12	10.5
4	13	11.2
5	14	12.0
6	16	13.6
7	17	14.5
8	18	15.5
9	19	16.5
10	20	17.5
11	21	18.7
12	22	19.8
13	23	21.1
14	24	22.4
15	26	25.2

Table 3-14 continued.

16	27	26.7
17	28	28.3
18	29	30.0
19	30	31.8
20	31	33.7
21	32	35.7
22	33	37.7
23	34	39.9
24	35	42.2

Table 3-15 shows the training results for the data in Table 3-14. After 3000 epochs, the objective function value (error) is 0.007. For the same manner as it was performed in case1 study, the neural network is also tested using the test data set in Table 3-16. Due to the limited available data, there are only two rows of the test data, but we still can see the applicability of the proposed algorithm. Table 3-17 presents the testing results for the test data in Table 3-16.

Table 3-15. Training results for the data in Table 3-14.

Sample Number	Temperature (^0C)	Pressure (mmHg) (experimental)	Pressure (mmHg) (computed)
1	10	9.2	9.2
2	11	9.8	9.8
3	12	10.5	10.5
4	13	11.2	11.2
5	14	12.0	12.0
6	16	13.6	13.6
7	17	14.5	14.5
8	18	15.5	15.5
9	19	16.5	16.5
10	20	17.5	17.5
11	21	18.7	18.7
12	22	19.8	19.8
13	23	21.1	21.1
14	24	22.4	22.4
15	26	25.2	25.2
16	27	26.7	26.7
17	28	28.3	28.3
18	29	30.0	30.0
19	30	31.8	
20	31	33.7	33.7

Table 3-15 continued.

21	32	35.7	35.7
22	33	37.7	37.7
23	34	39.9	39.8
24	35	42.2	42.2

Table 3-16. Test data for Water Vapor at Different Temperatures [39].

Sample Number	Temperature (^{0}C)	Pressure (mmHg)
1	15	12.8
2	25	23.8

Table 3-17. Testing Results for the Data in Table 3-16.

Sample Number	Temperature (^{0}C)	Pressure (mmHg) (experimental)	Pressure (mmHg) (computed)
1	15	12.8	12.8
2	25	23.8	23.8

CHAPTER IV CONCLUSIONS

The genetic algorithm is a robust method for global optimization. However, the traditional genetic algorithm requires a very large population size and is slow. The Chemotaxis method works on only one point, so its drawback is the possibility of being a trapped in a local minimum. The introduction of diversity in the genetic algorithm dramatically reduces the population size without loss of global optimization. Based on the ideas of the genetic algorithm with diversity guidance and the chemotaxis method, a novel algorithm, a simple genetic algorithm with diversity guidance combined with chemotaxis (GADCA) is proposed. From the case studies considered, GADCA demonstrates the following unique advantages as compared to the traditional genetic algorithm and the chemotaxis method.

1. Using the real variables in a certain range as weights simplifies the representation of the traditional bit-string coding system. This simplification makes the genetic algorithm easier to understand.
2. Introduction of Euclidean distance as the standard to keep a new point in a population is crucial for the diversity of the population. By this method, the promising candidates are always kept in the population for the next generation.
3. The Chemotaxis search method is adapted and combined with the genetic algorithm naturally.
4. GADCA should be fast for searching for a global minimum since it inherits both the advantages of the genetic algorithm and the chemotaxis method.

5. The drawback of the GADCA is the large amount of computing time for many iterations. Thus, there still exists room for further improvement of the proposed algorithm.

REFERENCES

[1] Holland, J. H., "Genetic Algorithms", *Scientific American*, July, 1992, pp. 66 – 87.

[2] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*; Addison-Wesley: Reading, MA, 1989.

[3] Bremermann, H. J. and Anderson, R. W., "An Alternative to Back Propagation: A Simple for Synaptic Modification for Neural Net Training and Memory", *Internal Report*, Dept. of Mathematics, University of California, Berkeley, 1989.

[4] Nelder, J. A. and Mead, R. A., "A Simplex Method for Function Minimization", Computer. J., 7, 1965, pp. 308 – 313.

[5] Masters, T., *Practical Neural Network Recipes in C++*, Academic Press, 1993.

[6] Holland, J. H., *Adaptation In Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[7] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag Berlin Heidelberg, 1996.

[8] Cormen, T. H., Charles, E. L. and Ronald, L. R., *Introduction to Algorithms*, McGraw-Hill, 1990.

[9] Davis, L., *Handbook of Genetic Algorithms*, Van Nostrand-Reinhold, 1991.

[10] Hibbert, D. A., "Genetic Algorithms in Chemistry", *Chemom. Intell. Lab. Syst.*, 19, 1993, pp. 277 – 293.

[11] Lucasius, C. B. and Kateman, G., "On K-medoid Clustering of Large Data Sets with the Aid of Genetic Algorithms: Background, Feasibility and Comparison", *Chemom. Intell. Lab. Syst.*, 25, 1994, pp. 99 – 145.

[12] Reeves, C. and Wright, C., "Genetic Algorithms and Statistical Methods: A Comparison", *First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, IEE Conference Publication No. 44, 1995, pp. 137 - 140.

[13] Adler, J., "Chemotaxis in Bacteria", *Science*, 166, 1969, pp. 1588 – 1597.

[14] MacNab, R. M. and Koshland, D. E., "The Gradient-sensing Mechanism in Bacterial Chemotaxis", *Proc. Nat. Acad. Sci.*, 69, 1972, pp. 2509 – 2512.

[15] Keller, E. F. and Segal, L. A., "Model for Chemotaxis", *J. Theor. Biol.*, 30, 1971, pp. 225 – 234.

[16] Rosen, G. "Fundamental Theoretical Aspects of Bacterial Chemotaxis", *J. Theor. Biol.*, 41, 1973, pp. 201 – 208.

[17] Ferrée T. C., Marcottee B. A., and Lockery S. R., "Neural Network Models of Chemotaxis in the Nematode Caenorhabditis Elegans", *Advances in Neural Information Processing Systems*, Vol. 9, The MIT Press, 1997, pp. 55.

[18] Web B. "Robots and Crickets and Ants: Models of Neural Control of Chemotaxis and Phonotaxis", *Neural Networks*, Vol. 11, No. 7-8, 1998, pp. 1479-1496.

[19] Koshland D. E., "Bacterial Chemotaxis as a Model Behavioral System, *Press Raven*, 1980.

[20] Ahmed M., T. Alkhamis, and M. Hasan, "Optimizing Discrete Stochastic Systems Using Simulated Annealing and Simulation", *Computers and Industrial Engineering*, 32, 1997, pp. 823-836.

[21] Okada M., S. Hara, S. Komaki, and N. Morinaga, "An Application of Simulated Annealing to the Design of Block Coded Modulation", *IEICE Transactions on Communications*, E79-b, 1, 1996, pp. 88-91.

[22] Park M., and Y. Kim, "A Systematic Procedure for Setting Parameters in Simulated Annealing Algorithms", *Computers and Operation Research*, 25, 1998, pp. 207-217.

[23] Karaboga D. and Pham D. T., *Intelligent Techniques*, Spring Verlag, May, 2000.

[24] Aarts E. and . Korst J., *Simulated Annealing Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, John Wiley & Son Ltd, January, 1989.

[25] Lee, L. D., "Gynandromorphs", *Master's Report*, Oklahoma State University, 1963.

[26] Romeijn H., and R. Smith, "Simulated Annealing for Constrained Optimization", *Journal of Global Optimization*, 5, 1994, pp. 101-126.

[27] Bos, M., Bos, A. and Van Der Linden, W. E., "Processing of Signals from An Ion-selective Electrode Array by A Neural Network", *Analytica Chemica Acta*, 233, 1990, pp. 31 – 39.

[28] Hornik, K., Stinchcombe, M. and White, H., "Mutilayer Feedforward Networks Are Universal Approximators", *Neural Networks*, 2, 1989, pp. 359 – 366.

[29] Barnard, E. and Casasent, D., "Shift Invariance and the Neocognitron", *Neural Networks*, 3, 1992, pp. 403 – 410.

[30] Gallant, R. and White, H., "On Learning the Derivatives of an Unknown Mapping with Multilayer Feedforward Networks", *Neural Networks*, 2, 1992, pp. 129 – 360.

[31] Maren, A., Harston, G. and Pap, R., *Handbook of Neural Computing Applications*, Academic Press, New York, 1990.

[32] Li, B. and Jiang, W., "An Efficient Algorithm for Complex Problems", *IEEE International Conference on Intelligent Processing Systems*, Oct. 1997, pp. 703-706.

[33] Carey, M. and Skiscim, C., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco, CA: Freeman, 1979.

[34] Bui, T. N. and Jones, C., "Finding Good Approximate Vertex and Edge Partitions is NP-Hard", *Information Processing Letters*, Vol. 42, 1992, pp. 153-159.

[35] Esbensen, H and Masumder, P., "SAGA: A Unification of the Genetic Algorithm with Simulated Annealing and its Application to Macro-Cell Placement", *Seventh International Conference on VLSI Design-January 1994*, pp. 211-314.

[36] Baughman, D. R. and Liu, Y. A., *Neural Networks in Bioprocessing and Chemical Engineering*, Academic Press, Inc., San Diego, California, 1995, pp. 218 – 219.

[37] Yang, R. and Douglas, I., "Simple Algorithm with Local Tuning: Efficient Global Optimizing Technique", *Journal Optimization Theory and Applications*, 98, 1998, pp. 449 – 465.

[38] Wang, L., "The Damped Newton Method: an ANN Learning Algorithm", *M.S.Thesis*, Computer Science Department, Oklahoma Sate University, 1995.

[39] Brenda, W. H., *Chemindustry Experiments*, University of Cincinnati, 1979.

APPENDIX A

PROGRAM LISTING

```
///////////////////////////////////////////////////////////////////////////////////////////////////
//                          Thesis Program                                        //
//   Title:       Simple Genetic Algorithm with Chemotaxis                        //
//                          Tunable Local Searching Optimization                  //
//   Name:        Li, Wei                                                         //
//   Institute:   Department of Computer Science                                  //
//                          Oklahoma Sate University                              //
//   Date:        January 8, 2000                                                 //
///////////////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////////////////////////
//This is a driving main. In this main, you have a variety                       //
//of choices such as training, testing, generalizing your neural                 //
//network. Meanwhile, the two methods used to generate random                    //
//number with original distribution between 0 and 1 are defined                  //
//in this class.                                                                 //
///////////////////////////////////////////////////////////////////////////////////////////////////
#include<iostream.h>
#include<stdlib.h>
#include<string.h>
#include<fstream.h>
#include<ctype.h>
#include<time.h>
#include<math.h>
#include "network.h"
void train();
void test();
void generalization();
void main(){
//==============================================================//
//This is program drive method. In this method, a couple of               //
//options are displayed on the screen for you to choose. They             //
//are training, testing and generalizing your network.                    //
//==============================================================//
        srand((unsigned)time(NULL));
        char tr_ts_gen;
```

```
        cout<<"This algorithm demonstrates perfomance of ";
        cout<<"a new method combining the genetic"<<endl;
        cout<<"algorithm with chemotaxis, which yeilds ";
        cout<<"better solution than either that of genetic"<<endl;
        cout<<" algorithm or that of chemotaxis algorithm."<<endl;
        cout<<" In this program, you have three choices, "<<endl;
        cout<<"they are pure genetic algorithm, ";
        cout<<"genetic algorithm with diversity,";
        cout<<"chemotaxis algorithm"<<endl;
        cout<<" and genetic algorithm with chemotaxis.";
        cout<<endl;
        cout<<"Enter 'T' to train the network:"<<endl;
        cout<<"Enter 'S' to test the network:"<<endl;
        cout<<"Enter 'G' to generalize the network:"<<endl;
        cout<<endl;
        cin>>tr_ts_gen;
        tr_ts_gen = toupper(tr_ts_gen);
        switch(tr_ts_gen){
         case 'T':
                cout<<"Now you are training the network."<<endl;
                train();
                break;
         case 'S':
                cout<<"Now you are testing the network."<<endl;
                test();
                break;
                case 'G':
                cout<<"Now you are generalizing ";
                cout<<"the network."<<endl;
                generalization();
                break;
                default:
                cout<<"Please enter T, S or G.";
        }
}
/////////////////////////////////////////////////////////////////////////////
//This train matod gives you the ways of training a neural          //
//network. For the simplicity, there is only one method             //
//implemented in this class.                                        //
/////////////////////////////////////////////////////////////////////////////
void train(){
//==========================================================//
//In this method, you are provided four options to train a          //
//neural network.  They are pure genetic algorithm, simple          //
//genetic algorithm with diversity, chemotaxis and simple           //
//genetic algorithm with chmotaxis.                                 //
```

```
//=================================================================//
        int choose;
        int i, check, pg, pc, num_of_points, iterations;
        network::get_layer_info();
        network::get_train_input();
        network::get_train_output();
        cout<<"Enter the number of points: "<<endl;
        cin>>num_of_points;
        GACA gaca(num_of_points);
        cout<<"1. pure genetic algorithm."<<endl;
        cout<<"2. genetic algorithm with diversity."<<endl;
        cout<<"3. chemotaxis algorithm."<<endl;
        cout<<"4. genetic algorithm with chemotaxis."<<endl;
        cin>>choose;
        switch(choose){
         case 1:
                cout<<"You have chosen pure genetic method."<<endl;
                cout<<"How many iterations do you want?"<<endl;
                cin>>iterations;
                for(i=0; i<iterations; i++){
                 check = gaca.checkPoints();
                        if(check==1){
                                cout<<"When all the points are the same,";
                                cout<<"the iteration number is: "<<(i+1)<<endl;;
                                break;
                        }
                 gaca.geneticAlgorithm(num_of_points);
                 gaca.nextGeneration();
                }
                gaca.getPoints().print_train();
                break;
         case 2:
                cout<<"You have chosen simple genetic method."<<endl;
                cout<<"How many iterations do you want?"<<endl;
                cin>>iterations;
                for(i=0; i<iterations; i++){
                 check = gaca.checkPoints();
                 if(check==1){
                        cout<<"When all the points are the same, ";
                        cout<<"the iteration number is: "<<(i+1)<<endl;
                        break;
                 }
                 gaca.geneticAlgorithm(num_of_points);
                 gaca.competition();
                 gaca.nextGeneration();
                }
```

```
            gaca.getPoints().print_train();
            break;
        case 3:
            cout<<"You have chosen chemotaxis method.";
            cout<<"Now, you are working on single point."<<endl;
            gaca.set_stepsize();
            cout<<"How many iterations do you want?"<<endl;
            cin>>iterations;
            for(i=0; i<iterations; i++){
             gaca.chemotaxis(0);
             gaca.competition();
             gaca.nextGeneration();
            }
            gaca.getPoints().print_train();
            break;
        case 4:
            cout<<"You have chosen genetic algrithm";
            cout<<" with chemotaxis method."<<endl;
            gaca.set_stepsize();
            cout<<"How many iterations do you want?"<<endl;
            cin>>iterations;
            for(i=0; i<iterations; i++){
                    check = gaca.checkPoints();
                    if(check==1){
                            cout<<"When all the points are the same ";
                            cout<<"the iteration number is: "<<(i+1)<<endl;
                            break;
                    }
                    pc = num_of_points * i/iterations;
                    pg = num_of_points - pc;
                    gaca.geneticAlgorithm(pg);
                    if(pc>=1){
                            gaca.combineChemo(pc);
                    }
                    gaca.competition();
                    gaca.nextGeneration();
            }
            gaca.getPoints().print_train();
            break;
            default:
             cout<<"You enetered an invalid number.";
             cout<<"Enter 1, 2 or 3."<<endl;
             exit(1);
        }
}//end of train
```

```
/////////////////////////////////////////////////////////////////////////////////////////
//The test method is implemented to get the test result with                          //
//a related error.                                                                    //
/////////////////////////////////////////////////////////////////////////////////////////
void test(){
//===================================================================//
//In this method, the test input files are put in and the                           //
//output files are also inputed. Meanwhile the minimum data                         //
//and the maximum data in the input training files are asked .                      //
//to normalize the input test files. The same way also is                           //
//for the output test files.                                                         //
//===================================================================//
        network::get_layer_info();
        network::get_test_input();
        network::get_test_output();
        GACA gaca(1);
        network net;
        net = gaca.getPoints();
        net.setup_weights();
        net.calc_output();
        net.calc_error();
        net.print_test();
}//end of test
/////////////////////////////////////////////////////////////////////////////////////////
//The class generalization is implemented to get a series of                         //
//output results. The generalization input files are inputed.                        //
//As illustrated in test class, the minimum and maximum data                         //
//in the train files are asked to input to normalized the                            //
//input files. However, in this class, there is no output                            //
//file.                                                                               //
/////////////////////////////////////////////////////////////////////////////////////////
void generalization(){
//===================================================================//
//In this method, the weights obtained from the training thc                          //
//neural network are inputed to set the neural network                               //
//parameters. The output results will be printed in a file.                          //
//===================================================================//
        network::get_layer_info();
        network::get_generalization_input();
        GACA gaca(1);
        network net;
        net = gaca.getPoints();
        net.setup_weights();
        net.calc_output();
        net.print_generalization();
```

```
}//end of generalization
////////////////////////////////////////////////////////////////////////////////////////////////////////
//Header file network.h                                                                              //
////////////////////////////////////////////////////////////////////////////////////////////////////////
#include<iostream.h>
#include<stdlib.h>
#include<math.h>
#include<ctype.h>
#include<string.h>
#include<time.h>
//==============================================================//
//This method is implemented to generate a series of random              //
//weights for the neural network. These random weights are in a          //
//certain range of -20 and 10. The loop is used to avoid creating        //
//the same points with identical weights.                                //
//==============================================================//
double random_weights(){
  double rdNum;
      for(int i=0; i<10; i++)
            rdNum = double(rand()/32767.0);
      //convert the random number to a value between -20 and 10
      rdNum = -20.0 + rdNum * (10.0 - (-20.0));
      return rdNum;
 }//end of random_weights
//==============================================================//
//This method is used to just generate a random number between 0         //
//and 1. For the same purpose as above method, the loop is used          //
//to avid creating two identical random numbers.                         //
//==============================================================//
double random_generator(){
  double rdNum;
      for(int i=0; i<10; i++)
       rdNum = double(rand()/32767.0);
      return rdNum;
 }//end of random_generator
////////////////////////////////////////////////////////////////////////////////////////////////////////
//Class network represents a complete neural network                     //
//structure. In this class, primarily, a set of randomly                 //
//generated numbers are used to set up the weights for the               //
//neural network. A series of method implementaions are                  //
//defined in this class. Their functionalities will be                   //
//discussed when they are implemented.                                   //
////////////////////////////////////////////////////////////////////////////////////////////////////////
class network {
private:
 //array for a number fo layers
```

```
 static int numLayer[5];
 //number of layers of the neural network
 static int numOfLayers;
 //number of the weights in the neural network
 static int numOfWeights;
 //array of weights stores the individual weight for
 //the neural network
 double weights[50];
 //number of the input data inputed
 static int numOfInputData;
 //array to store the input data
 static double inputData[5][90];
 //array to store normalized input data
 static double normInputData[5][90];
 //number of output data
 static int numOfOutputData;
 //array to store output data
 static double outputData[5][90];
 //array to store normalized output data
 static double normOutputData[5][90];
 //array to store final output computed by the neural
 //network
 double finalOutput[5][90];
 friend class GACA;
public:
        network();
        static void get_layer_info();
        void initialize_weights();
        void setup_weights();
        static void get_train_input();
        static void get_test_input();
        static void get_generalization_input();
        static void get_train_output();
        static void get_test_output();
        void calc_output();
        void get_first_layer_input(double tmp1[], int nm);
        int calc_temp_out(double tmp1[], double tmp2[],
                          int nr1, int nr2, int count, int last);
        double segmoid(double val);
        void calc_error();
        void print_train();
        void print_test();
        void print_generalization();
        int get_num_weights();
};
int network::numLayer[5];
```

```
int network::numOfLayers = 0;
double network::inputData[5][90];
double network::normInputData[5][90];
int network::numOfOutputData = 0;
double network::outputData[5][90];
double network::normOutputData[5][90];
int network::numOfWeights = 0;
int network::numOfInputData = 0;
//==============================================================//
//Default constructor for the class network.                    //
//==============================================================//
network::network(){ }
//==============================================================//
//The method is used to get input files and normalized the      //
//the input data in the range of 0 and 1 to avoid               //
//saturation of the data.                                       //
//==============================================================//
void network::get_layer_info(){
        int i=0;
        cout<<"Enter the number of";
        cout<<"layers for the neural network:"<<endl;;
        cin>>numOfLayers;
        //put the number of neurons in the different layers
        //into array of numLayer
        while(i<numOfLayers){
          cout<<"Enter the number of neurons in ";
          cout<<"layer" <<(i+1)<<":"<<endl;
          cin>>numLayer[i];
          i++;
        }
        i = 0;
        while(i<numOfLayers-1){
          numOfWeights = numOfWeights+numLayer[i]*numLayer[i+1]
                    +numLayer[i+1];
          i++;
        }
        //two additional numbers of weight are for error
        //and probability
        numOfWeights = numOfWeights + 2;
}//end of get_layer_info
//==============================================================//
//In this method, the total number of weights in the neural     //
//network is calculated and randomly generated weight is put    //
//the array of weights. Also the array of finalOutput is        //
//created for storing the output computed by the neural         //
//network.                                                      //
```

```
//==============================================================//
void network::initialize_weights(){
        int i=0;
        //put the random number in the array of weights
        for(i=0; i<numOfWeights-2; i++)
          weights[i] = random_weights();
}//end of initialize_weights
//==============================================================//
//This method is used for testing or generalizing the neural       //
//network. The weights obtained from training the neural          //
//network are kept in a input file. Then the weights in the        //
//input file are stored in the array of weights.                  //
//==============================================================//
void network::setup_weights(){
        char filename[80];
        ifstream infile;
        double data;
        int i=0;
        numOfWeights = 0;
        //calculate the total number of weights
        while(i<numOfLayers-1){
                numOfWeights = numOfWeights+
                        numLayer[i]*numLayer[i+1]+numLayer[i+1];
                i++;
        }
        numOfWeights = numOfWeights + 2;
        cout<<"Enter the weight file name:"<<endl;
        cin>>filename;
        infile.open(filename, ios::in|ios::nocreate);
        i = 0;
        while(!infile.eof()){
                infile>>data;
                weights[i]=data;
                i++;
        }
        infile.close();
}//end of setup_weights
//==============================================================//
//The input data is stored in an array of inputData, then the     //
//data in the array is normalized and put in the array of         //
//normInputData                                                   //
//==============================================================//
void network::get_train_input(){
        ifstream infile;
        double data, min, max;
        char filename[80];
```

```
	int i=0, j;
	while(i<numLayer[0]){
		cout<<"Enter the train input ";
		cout<<"file "<<(i+1)<<" name: "<<endl;;
		cin>>filename;
		//open the input data files
		infile.open(filename, ios::in|ios::nocreate);
		numOfInputData=0;
		while(!infile.eof()){
			infile>>data;
			//store the data in an array
			inputData[i][numOfInputData]=data;
			numOfInputData++;
		}
		infile.close();
		//find the minimum and maximum for normalizing
		//the input data
		min = max = inputData[i][0];
		for(j=1; j<numOfInputData; j++){
		 if(inputData[i][j]<min)
			min=inputData[i][j];
		 if(inputData[i][j]>max)
			max=inputData[i][j];
		}
		//normalize the input data and put it in an array
		//of normInputData
		for(j=0; j<numOfInputData; j++)
		 normInputData[i][j] = 1.0/(max-min)*
		   (inputData[i][j]-min);
		normInputData[i][j] = min;
		normInputData[i][j+1] = max;
		i++;
	}
 }//end of get_train_input
//=============================================================//
//The input test file is opened and stored in an array of        //
//inputData. Then the data in the array is normalized and        //
//put in an array of normInputData.                              //
//=============================================================//
void network::get_test_input(){
	ifstream infile;
	double data, min, max;
	char filename[80];
	int i=0, j;
	while(i<numLayer[0]){
	 cout<<"Enter the test input file name: "<<(i+1)<<endl;;
```

```
		cin>>filename;
		//open the input test file
		infile.open(filename, ios::in|ios::nocreate);
		numOfInputData=0;
		while(!infile.eof()){
			infile>>data;
			//put the data in the test file in
			//an array of inputData
			inputData[i][numOfInputData]=data;
			numOfInputData++;
		}
		infile.close();
		//get the minimun and maximum data in the
		//corresponding training input file
		cout<<"Enter the min data ";
		cout<<"in the train data set:"<<endl;
		cin>>min;
		cout<<"Enter the max data ";
		cout<<"in the train data set:"<<endl;
		cin>>max;
		//normalize the test data
		for(j=0; j<numOfInputData; j++)
			normInputData[i][j]=1.0/(max-min)*(inputData[i][j]-min);
		normInputData[i][j]=min;
		normInputData[i][j+1]=max;
		i++;
	}
}//end of get_test_input
//==============================================================//
//The generalization input test file is opened and stored in      //
//an array of inputData. Then the data in the array is            //
//normalized and put in an array of normInputData.                //
//==============================================================//
void network::get_generalization_input(){
 ifstream infile;
 double data, min, max;
 char filename[80];
 int i=0, j;
 while(i<numLayer[0]){
	cout<<"Enter the generalization input file name: "<<(i+1)<<endl;
	cin>>filename;
	infile.open(filename, ios::in|ios::nocreate);
	numOfInputData=0;
	while(!infile.eof()){
		infile>>data;
		inputData[i][numOfInputData]=data;
```

```
                numOfInputData++;
        }
        infile.close();
        cout<<"Enter the min in the train data set:"<<endl;
        cin>>min;
        cout<<"Enter the max in the train data set:"<<endl;
        cin>>max;
        for(j=0; j<numOfInputData; j++)
          normInputData[i][j]=1.0/(max-min)*(inputData[i][j]-min);
        normInputData[i][j]=min;
        normInputData[i][j+1]=max;
        i++;
        }
  }//end of get_generalization_input
//===========================================================//
//The input data is stored in an array of outputData, then       //
//the data in the array is normalized and put in the array        //
//of normOutputData                                                 //
//===========================================================//
void network::get_train_output(){
        ifstream infile;
        double data, min, max;
        char filename[80];
        int i=0, j;
        while(i<numLayer[numOfLayers-1]){
          cout<<"Enter the output file name: "<<(i+1)<<endl;
          cin>>filename;
          //open file
          infile.open(filename, ios::in|ios::nocreate);
          while(!infile.eof()){
                infile>>data;
                //put the data into an array
                outputData[i][numOfOutputData]=data;
                numOfOutputData++;
          }
          infile.close();
          //find the minimum and maximum in the output
          //file
          min = max = outputData[i][0];
          for(j=1; j<numOfOutputData; j++){
                if(outputData[i][j]<min)
                  min=outputData[i][j];
                if(outputData[i][j]>max)
                  max=outputData[i][j];
          }
          //normalize the data
```

```
        for(j=0; j<numOfOutputData; j++)
              normOutputData[i][j]=1.0/(max-min)*
                (outputData[i][j]-min);
        normOutputData[i][j]=min;
        normOutputData[i][j+1]=max;
        i++;
      }
}//end of get_train_output

//=============================================================//
//The output test file is opened and stored in an array of      //
//outputData. Then the data in the array is normalized and      //
//put in an array of normOutputData.                            //
//=============================================================//
void network::get_test_output(){
      ifstream infile;
      double data, min, max;
      char filename[80];
      int i=0, j;
      while(i<numLayer[numOfLayers-1]){
        cout<<"Enter the output file name: "<<(i+1)<<endl;
        cin>>filename;
        //open file
        infile.open(filename, ios::in|ios::nocreate);
        numOfOutputData=0;
        while(!infile.eof()){
              infile>>data;
              //put data into an array
              outputData[i][numOfOutputData]=data;
              numOfOutputData++;
        }
        infile.close();
        //get the minimum and maximum data in the
        //corresponding training file
        cout<<"Enter the min in the train ";
        cout<<"output data set:"<<endl;
        cin>>min;
        cout<<"Enter the max in the train";
        cout<<" output data set:"<<endl;
        cin>>max;
        //normalize the data
        for(j=0; j<numOfOutputData; j++)
              normOutputData[i][j]=1.0/(max-min)*
                (outputData[i][j]-min);
              normOutputData[i][j]=min;
              normOutputData[i][j+1]=max;
```

```
            i++;
        }
}//end of get_test_output
//===========================================================//
//This method is used to calculate the output of the neural          //
//network.  Two other method are called for the                     //
//calculations.  Their functionalities will be discussed            //
//later.                                                              //
//===========================================================//
void network::calc_output(){
        int h, i, j, k, m, n, weight_count, last_layer;
        n=0;
        i=0;
        h=0;
        k=0;
        //array temps are used to store temperaly data
        //calculated by the neural network
        double temp1[20], temp2[20];
        while(k<numOfInputData){
          weight_count=0;
          //put the data in the first layer into the array
          //of temp1
          get_first_layer_input(temp1, k);
                for(j=1; j<numOfLayers; j++){
                  last_layer=j;
                  //calculated the output in the consecutive layer
                  //and put them into array of temp2
                  weight_count=calc_temp_out(temp1, temp2, numLayer[j-1],
                        numLayer[j], weight_count, last_layer);
                  //copy the data in temp2 into temp1
                  //for next layer calculation
                  for(m=0; m<numLayer[j]; m++)
                        temp1[m]=temp2[m];
                }
                i++;
                //put final result computed by the neural network
                //into array of finalOutput
                for(n=0; n<numLayer[numOfLayers-1]; n++){
                  finalOutput[n][k]=temp1[n];
                }
                k++;
        }
}//end of calc_output
//===========================================================//
//This helper method is used to get the data in the                 //
//normInputData in a temperary array of tmp1.                       //
```

```
//==========================================================//
void network::get_first_layer_input(double tmp1[], int nm){
        int i;
        for(i=0; i<numLayer[0]; i++)
          tmp1[i] = normInputData[i][nm];
 }//end of get_first_layer_input
//==========================================================//
//This helper method is used to calculate the consecutive        //
//layer output and put it into a temporary array of tmp2.        //
//==========================================================//
int network::calc_temp_out(double tmp1[], double tmp2[],
                        int nr1, int nr2, int count, int last){
        int i, j;
        double value;
        for(i=0; i<nr2; i++){
          value=0.0;
          //calculate the output in a consecutive layer
          for(j=0; j<nr1; j++){
                value+=tmp1[j]*weights[count];
                count++;
          }
          value = value+weights[count];
          count++;
          //this output should be between 0 and 1
          if(last!=numOfLayers-1)
                tmp2[i]=segmoid(value);
          //last layer should be linear without using
          //segmoid function
          else
                tmp2[i]=value;
        }
        return count;
}//end of calc_temp_out
//==========================================================//
//This helper method is to get the output in a consecutive       //
//layer in the neural network.                                   //
//==========================================================//
double network::segmoid(double val){
 double output;
 output = 1/(1+exp(-val));
 return output;
}//end of segmoid
//==========================================================//
//This method calculates the error between the computed          //
//results by the neural network and the result from the          //
//the input file.                                                //
```

```
//==============================================================//
void network::calc_error(){
 int i, j;
 double sum=0.0;
 double stdDev;
 //calculate the error
 for(i=0; i<numLayer[numOfLayers-1]; i++)
        for(j=0; j<numOfOutputData; j++)
          sum += (normOutputData[i][j]-finalOutput[i][j])*
                 (normOutputData[i][j]-finalOutput[i][j]);
        stdDev = sqrt(sum);
        weights[numOfWeights-2]=stdDev;
}//end of calc_error
//==============================================================//
//This method is implemented to denormalize the final data      //
//computed by the neural network. The denormalized adta         //
//is printed out.                                               //
//==============================================================//
void network::print_train(){
 ofstream outfile;
 int i, j;
 double data;
 //open an output file
 outfile.open("train.dat", ios::out);
 for(i=0; i<numLayer[numOfLayers-1]; i++){
          for(j=0; j<numOfOutputData; j++){
                 //denormalize the data
                 data = (normOutputData[i][numOfOutputData+1]-
                   normOutputData[i][numOfOutputData])*
                   finalOutput[i][j]+normOutputData[i]
                   [numOfOutputData];
                  outfile<<data<<" ";
           }//end_for
 }//end_for
 outfile<<endl;
 //print out weights
 outfile<<"The weights are: "<<endl;
 for(i=0; i<numOfWeights-2; i++){
        data=weights[i];
        outfile<<data<<" ";
  }
 //print out error
 outfile<<endl;
 outfile<<"The standard deviation is:"<<endl;
 data = weights[numOfWeights-2];
 outfile<<data;
```

```
 outfile.close();
}//end of print_train
//======================================================//
//This method is implemented to denormalize the final data        //
//computed by the neural network for the test. The                //
//denormalized adta is printed out.                               //
//======================================================//
void network::print_test(){
 ofstream outfile;
 int i, j;
 double data;
 //open output file
 outfile.open("test.dat", ios::out);
 outfile<<"The final outputs are: "<<endl;
 for(i=0; i<numLayer[numOfLayers-1]; i++){
        for(j=0; j<numOfOutputData; j++){
         data = (normOutputData[i][numOfOutputData+1]-
                normOutputData[i][numOfOutputData])*
                finalOutput[i][j]+normOutputData[i]
                [numOfOutputData];
         outfile<<data<<" ";
        }
 }
 //print the error
 outfile<<endl;
 outfile<<"The standard deviation is: "<<endl;
 data = weights[numOfWeights-2];
 outfile<<data<<endl;
 outfile.close();
}//end of print_test
//======================================================//
//This method is implemented to denormalize the final data        //
//computed by the neural network for the generalization.          //
//The the denormalized data is printed out.                       //
//======================================================//
void network::print_generalization(){
 ofstream outfile;
 int i, j;
 double data, max, min;
 outfile.open("gen.dat", ios::out);
 //get the minimum and maximum data in the
 //corresponding train file
 cout<<"Enter the min in the ";
 cout<<"train output data set:"<<endl;
 cin>>min;
 cout<<"Enter the max in the ";
```

```
 cout<<"train output data set:"<<endl;
 cin>>max;
 //open output file
 outfile<<"The final outputs are: "<<endl;
 //denormalize the data
 for(i=0; i<numLayer[numOfLayers-1]; i++){
        for(j=0; j<numOfInputData; j++){
          data = (max-min)*finalOutput[i][j]+ min;
          outfile<<data<<" ";
        }
 }
 outfile.close();
}//end of print_generalization
//=========================================================//
//The method is implemented to get the number of total           //
//weights in a neural network.                                   //
//=========================================================//
int network::get_num_weights(){
   return numOfWeights;
}//end of get_num_weights
/////////////////////////////////////////////////////////////////////////////
//Class GACA is to generate a couple of networks called              //
//points.  In this class, the all algorithms will be                  //
//performed.  For individual algorithm, it will be                    //
//discussed when it is implemented.                                   //
/////////////////////////////////////////////////////////////////////////////
class GACA{
private:
 //a couple of points
 network ntwk[100];
 //a couple new points
 network newNtwk[100];
 //number of points
 int numOfPoints;
 //two points
 network twoPoints[2];
 //one point
 network cross;
 //another one point
 network chemo;
 //for chemotaxis
 double stepSize;
public:
 GACA(int points);
 void heapsort(network net[], int size);
 void heapify(network net[], int pos, int size);
```

```
  void assign_probability();
  void select_two_points();
  int get_index(double rd);
  void crossover();
  void mutation();
  void geneticAlgorithm(int pg);
  void combineChemo(int pc);
  void competition();
  double distance(network net[], int indexOfPoints);
  void set_stepsize();
  void chemotaxis(int pc);
  void nextGeneration();
  int checkPoints();
  network getPoints();
};
//==========================================================//
//This constructor is for test and generalization, since     //
//in this situation, only one point is needed.               //
//==========================================================//
GACA::GACA(int points){
        int i;
        numOfPoints = points;
        //get the weights, calculated output
        //and eeror for each point
        //Random number = new Random();
        for(i=0; i<numOfPoints; i++){
          ntwk[i].initialize_weights();
          ntwk[i].calc_output();
          ntwk[i].calc_error();
        }
        //sort the points
        heapsort(ntwk, numOfPoints);
        //assign probability for each point
        assign_probability();
}//end of constructor
//========================================================//
//This helper method is to sort a couple of points         //
//according to their error in ascending order.             //
//========================================================//
void GACA::heapsort(network net[], int size){
        network temp;
        int i, j;
        for(i=(size-1)/2; i>=0; i--)
          heapify(net, i, size);
        for(i=size-1; i>0; i--){
          for(j=0; j<temp.get_num_weights()-1; j++){
```

```
            temp.weights[j] = net[0].weights[j];
            net[0].weights[j] = net[i].weights[j];
            net[i].weights[j] = temp.weights[j];
       }
       heapify(net, 0, i);
      }
}//end of heapsort
//==========================================================//
//This helper method is to help to sort the couple of       //
//points.                                                   //
//==========================================================//
void GACA::heapify(network net[], int pos, int size){
      int j, l, r, k, largest;
      network temp;
      j = pos;
      while(j<size-1){
       l = 2*j;
       r = 2*j+1;
       if(l<=size-1){
            if(net[l].weights[temp.get_num_weights()-2]>
             net[j].weights[temp.get_num_weights()-2])
             largest = l;
            else
             largest = j;
            }
            if(r<=size-1){
             if(net[r].weights[temp.get_num_weights()-2]>
                  net[largest].weights[temp.get_num_weights()-2])
                  largest = r;
            }
            if(largest!=j){
             for(k=0; k<temp.get_num_weights()-1; k++){
                  temp.weights[k] = net[j].weights[k];
                  net[j].weights[k] = net[largest].weights[k];
                  net[largest].weights[k] = temp.weights[k];
             }
             j = largest;
            }
            else
             break;
            }
}//end of heapify
//==========================================================//
//This method is implemented to assign probability to each  //
//point according to its error.                             //
//==========================================================//
```

```
void GACA::assign_probability(){
        network temp;
        int i;
        double c, pm, pb, p;
        c = 0.5;
        pm = c/(double)numOfPoints;
        pb = (2.0-c)/(double)numOfPoints;
        //assign the probability
        for(i=numOfPoints; i>0; i--){
              p = pm+((double)(i-1)/(double)(numOfPoints-1))*
               (pb-pm);
              ntwk[numOfPoints-i].weights[temp.get_num_weights()-1]
               = p;
              }
}//end of assign_probability
//=============================================================//
//This method is to find two different points among the       //
//original population.                                         //
//=============================================================//
void GACA::select_two_points(){
        network temp;
        int i, j, index, flag;
        double rdNum;
        flag = 1;
        //select two different points
        while(flag==1){
              for(i=0; i<2; i++){
               rdNum = random_generator();
               index = get_index(rdNum);
               for(j=0; j<temp.get_num_weights()-2; j++)
                     twoPoints[i].weights[j] = ntwk[index].weights[j];
              }
         //check whether the two points are the same
         for(i=0; i<temp.get_num_weights()-2; i++)
               if(twoPoints[0].weights[i]!=twoPoints[1].weights[i]){
                flag = 0;
                break;
               }
        }
}//end of select_two_points
//======================================================//
//This helper method is to return the index of point in //
//the population according to the random generated      //
//number.                                               //
//======================================================//
int GACA::get_index(double rd){
```

```
        network temp;
        double start, end;
        int i, target_index=0;
        start = end = 0.0;
        if(rd<ntwk[0].weights[temp.get_num_weights()-1])
                target_index = 0;
        //calculate the probabilty range
        for(i=1; i<numOfPoints; i++){
                start += ntwk[i-1].weights[temp.get_num_weights()-1];
                end = start + ntwk[i].weights[temp.get_num_weights()-1];
                if(start<rd&&rd<=end)
                        target_index = i;
                }
                return target_index;
}//end of get_index
//=========================================================//
//This helpre method is to get one point from the          //
//selected two points.                                     //
//=========================================================//
void GACA::crossover(){
        int i;
        network temp;
        double rdNum;
        //get one point from the selected two points
        for(i=0; i<temp.get_num_weights()-2; i++){
                rdNum = random_generator();
                if(rdNum >= 0.5)
                        cross.weights[i] = twoPoints[0].weights[i];
                else
                        cross.weights[i] = twoPoints[1].weights[i];
                }
}//end of crossover
//=========================================================//
//This helper method is to change the obtained one point   //
//according to the randomly generated number.              //
//=========================================================//
void GACA::mutation(){
        network temp;
        double rdNum;
        int i;
        for(i=0; i<temp.get_num_weights()-2; i++){
                rdNum = random_generator();
                 //change the weight according to the condition
                if(rdNum < 0.05)
                        cross.weights[i] = random_weights();
        }
```

```
}//end of mutation
//============================================================//
//In this method, the new population is generated according          //
//to the operators of the genetic algorithm.                         //
//============================================================//
void GACA::geneticAlgorithm(int pg){
        network temp;
        int i, j;
        for(i=0; i<pg; i++){
                select_two_points();
                crossover();
                mutation();
                //copy the point after mutation
                for(j=0; j<temp.get_num_weights()-2; j++)
                  newNtwk[i].weights[j] = cross.weights[j];
        }
}//end of geneticAlgorithm
//=========================================================//
//This method is to manipulate the chemotaxis under genetic algorithm.   //
//=========================================================//
void GACA::combineChemo(int pc){
        int i;
        for(i=(numOfPoints-pc); i<numOfPoints; i++)
                chemotaxis(i);
}//end of combineChemo
//=====================================================//
//The two populations are competed in this method.             //
//The result of the competition is that the best-so-           //
//far point is kept.                                           //
//=====================================================//
void GACA::competition(){
        network temp;
        int i, j;
        double rdNum;
        for(i=0; i<numOfPoints; i++){
                newNtwk[i].calc_output();
                newNtwk[i].calc_error();
        }
        heapsort(newNtwk, numOfPoints);
        //check the best in the new points and old points
        if(newNtwk[0].weights[temp.get_num_weights()-2]>
                ntwk[0].weights[temp.get_num_weights()-2]){
        //replace the worst point in the new points
        //by the best point in the old points
        for(i=0; i<temp.get_num_weights()-1; i++)
                newNtwk[numOfPoints-1].weights[i]=ntwk[0].weights[i];
```

```
            heapsort(newNtwk, numOfPoints);
        }
        //check other points
        for(i=1; i<numOfPoints; i++){
            if((newNtwk[i].weights[temp.get_num_weights()-2]<
             ntwk[i].weights[temp.get_num_weights()-2])&&
             (distance(newNtwk, i)>distance(ntwk, i))){
             //keep the point
                ;
          }
            else if((distance(ntwk, i)>distance(newNtwk, i))&&
             (ntwk[i].weights[temp.get_num_weights()-2]<
             newNtwk[i].weights[temp.get_num_weights()-2])){
            //replace the new point by the cprresponding point
             for(j=0; j<temp.get_num_weights()-1; j++)
                newNtwk[i].weights[j] = ntwk[i].weights[j];
             heapsort(newNtwk, numOfPoints);
             }
             else if(distance(newNtwk, i)*
                ntwk[i].weights[temp.get_num_weights()-2]>
                distance(ntwk, i)*
                newNtwk[i].weights[temp.get_num_weights()-2]){
                ;
             }
             else{
                /* for(j=0; j<temp.get_num_weights()-2; j++)
                    newNtwk[i].weights[j] = op.random_weights(number);
                 newNtwk[i].calc_output();
                 newNtwk[i].calc_error();
                 heapsort(newNtwk, numOfPoints);*/
              rdNum = random_generator();
              if(rdNum>0.5){
                 for(j=0; j<temp.get_num_weights()-1; j++)
                    newNtwk[i].weights[j] = ntwk[i].weights[j];
                 heapsort(newNtwk, numOfPoints);
                }
                else
                 ;
             }//end_else
        }
}//end of competition
//=======================================================//
//This distance between the point and the best-so-far          //
//point is calculated in this method.                          //
//=======================================================//
double GACA::distance(network net[], int indexOfPoints){
```

```
	network temp;
	double d;
	int i;
	d = 0.0;
	for(i=0; i<temp.get_num_weights()-2; i++)
		d += (net[indexOfPoints].weights[i]-
		  newNtwk[0].weights[i])*(net[indexOfPoints].weights[i]
		  -newNtwk[0].weights[i]);
		return sqrt(d);
}//end of distance
//===========================================================//
//When the algorithm is chemotaxis, a step size is required.       //
//The step size is asked to input in this method.                  //
//===========================================================//
void GACA::set_stepsize(){
	cout<<"Enter the step size ";
	cout<<"for the chemotaxis method:"<<endl;
	cin>>stepSize;
}//end of set_stepsize
//========================================================//
//The chemotaxis algorithm is carried out in this method.       //
//========================================================//
void GACA::chemotaxis(int pc){
	int i, flag, less;
	network temp;
	flag = 1;
	// Random number = new Random();
	while(flag<100){
		less = 1;
		chemo.initialize_weights();
		while(less==1){
			//time the step size and added to the best point
			for(i=0; i<temp.get_num_weights()-2; i++){
				temp.weights[i] = chemo.weights[i]*stepSize;
				temp.weights[i] = ntwk[0].weights[i] +
					temp.weights[i];
			}
			//calculate the error
			temp.calc_output();
			temp.calc_error();
			if(temp.weights[temp.get_num_weights()-2]<
				ntwk[0].weights[temp.get_num_weights()-2]){
				less = 1;
			}
			else
				less = 0;
```

```
                if(less==1){
                        for(i=0; i<temp.get_num_weights()-1; i++)
                                newNtwk[pc].weights[i] = temp.weights[i];
                        //competition();
                        nextGeneration();
                        //flag = 0;
                }
            }//end while(less==1)
            flag++;
        }//end while(flag==1)
        //replace the worst point point by the bestPoint
       /* for(i=0; i<temp.get_num_weights()-1; i++)
                newNtwk[pc].weights[i] = chemo.weights[i];*/
}//end of chemotaxis
//==============================================//
//In this method teh new population is copied        //
//to the old population and the error is             //
//calculated.  the obtained old population is        //
//ready for next generation.                         //
//==============================================//
void GACA::nextGeneration(){
        network temp;
        int i, j;
        for(i=0; i<numOfPoints; i++){
                newNtwk[i].calc_output();
                newNtwk[i].calc_error();
        }
        heapsort(newNtwk, numOfPoints);
        //copy the newNtwk into ntwk for next generation
   for(i=0; i<numOfPoints; i++)
                for(j=0; j<temp.get_num_weights()-1; j++)
                  ntwk[i].weights[j] = newNtwk[i].weights[j];
        for(i=0; i<numOfPoints; i++){
                ntwk[i].calc_output();
                ntwk[i].calc_error();
        }
        heapsort(ntwk, numOfPoints);
        assign_probability();
}//end of nextGeneration
//=====================================================//
//This method is to check the points whether they are equal or not.   //
//=====================================================//
int GACA::checkPoints(){
        int i, j, repeat=0;
        int totalWeights=0;
        network temp;
```

```
	totalWeights = (numOfPoints-1)*(temp.get_num_weights()-2);
	for(i=1; i<numOfPoints; i++)
		for(j=0; j<temp.get_num_weights()-2; j++)
			if(ntwk[0].weights[j]==ntwk[i].weights[j])
				repeat++;
	if(repeat==totalWeights)
		return 1;
	else
		return 0;
}//end of checkPoints
//==========================================================//
//This method is to get the wanted point.                   //
//==========================================================//
network GACA::getPoints(){
	return ntwk[0];
}
```

VITA

Wei Li

Candidate for the Degree of Master of Science

Thesis: Genetic Algorithm with Chemotaxis Tunable Local Searching Optimization

Major Field: Computer Science

Education: Graduated from Changchun University of Science and Technology, Changchun, Jilin province, P. R. China in January 1982; received Bachelor of Science degree in Chemistry. Then graduated from Murray State University, Murray, Kentucky, USA in December 1993; received Master of Science degree in Engineering Technology. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May, 2001.

Experience: Employed as an engineer by Zhejiang Geological Institute from 1982 to 1990. Worked as a chemist in LCU Institute of Water Research in Lubbock, Texas from 1994 to 1996.