1991

Machine function identification system based on genetic algorithms

Mingda Jiang
The University of Montana

A MACHINE FUNCTION IDENTIFICATION SYSTEM BASED ON GENETIC ALGORITHMS

by

Mingda Jiang

B.S., Northeast Heavy Machinery Institute, 1982

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

in Computer Science

University of Montana

1991

Approved by

[illegible]

Chairman, Thesis Committee

[illegible]

Dean, Graduate School

Nov. 12, 1991

Date

Jiang, Mingda, MS, November 1991 Computer Science

A Machine Function Identification System Based on Genetic Algorithms (99 pp.)

Director: Alden H. Wright [illegible]

The function identification problem is a fundamental problem of science, medicine and engineering. The function identification problem is to find a function model of a system, in symbolic form, that fits the experimental data points of the system. In this problem, the fundamental properties of a system are to be determined from observed behavior of that system. The function model is a mathematical idealization that is used as an approximation to represent the output of the system. It is "good fit" to the given experimental data according to a chosen evaluation criterion.

This thesis is concerned with the construction of a general intelligent machine system to solve function identification problems. The machine function identification system does not need any priori knowledge about the system or function model to find a function, in symbolic form, that fits a set of given sample data points. The author combines a symbolic computing method and a numeric computing method to identify function models. The new approach dynamically creates a highly fit function model to the given sample data points, using Darwinian principles of reproduction and survival of the fittest, and optimizes the coefficients of the function model using the nonlinear regression algorithm. To this end, a machine function identification system, HGSFI, is implemented.

As a demonstration of the feasibility of the design an HGSFI implementation is tested in two categories of function identification problems: linear regression problems and nonlinear regression problems. Initialized in each test with randomly generated initial function models, the HGSFI implementation is shown to rapidly converge to high performance function models in both task domains.

ACKNOWLEDGMENTS

I would like to express my thanks to those who have helped make this research possible. A special note of appreciation to Alden H. Wright, my thesis advisor, for providing a constant source of guidance and new perspectives, to my thesis committee members Youlu Zheng and David Patterson for their enthusiasm about my thesis research, and especially to my wife Changhui Zhao who was called on in countless capacities and always responded with unwavering support, understanding, and encouragement.

The computer experiments were run on the excellent facilities provided by the University of Montana Computer Center.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iii

1. INTRODUCTION TO THE PROBLEM 1

1.1. INTRODUCTION 1

1.2. SOME RELATED EFFORTS 2

1.3. DISCUSSION 6

2. HGSFI -- A MACHINE FUNCTION IDENTIFICATION SYSTEM 9

2.1. INTRODUCTION 9

2.2. BACKGROUND 10

2.2.1. GENETIC ALGORITHMS 10

2.2.1.1. BASIC GENETIC ALGORITHM 12

2.2.1.2. HIERARCHICAL GENETIC ALGORITHM 18

2.2.2. NONLINEAR REGRESSION 25

2.3. THE KNOWLEDGE REPRESENTATION OF HGSFI 33

2.4. THE FITNESS FUNCTION 34

2.5. THE LEARNING PARADIGM OF HGSFI 38

3. THE IMPLEMENTATION OF HGSFI 43

3.1. DATA STRUCTURES 44

3.1.1. INDIVIDUAL AND NODE 44

3.1.2. POPULATION 46

3.1.3. GENETIC_SYSTEM 48

3.2. THE FRAMEWORK OF THE MAIN PROGRAM 52

3.3. THE SELECTION OF CONTROL PARAMETER VALUES 61

4. TESTING AND EVALUATION 68

4.1. COMPARISON OF THE SYSTEM PERFORMANCE WITH AND WITHOUT LEVENBERG-MARQUARDT OPTIMIZATION 69

4.2. LINEAR REGRESSION MODEL PROBLEMS 78

4.3. NONLINEAR REGRESSION MODEL PROBLEMS 83

5. FUTURE RESEARCH 90

6. SUMMARY AND CONCLUSIONS 93

BIBLIOGRAPHY 96

1. INTRODUCTION TO THE PROBLEM

1.1. INTRODUCTION

One of the aims of sciences is to find, to describe, and to predict relationships among events in the world in which we live. One way that this is accomplished is by finding a formula or equation that relates quantities in the real world. We may be interested, for example, in the relationship between temperature and pressure in a chemical process; or in the relationship between the number of apples on trees in an orchard and the amount of fertilizer the trees receive; or we may be interested in the relationship of supply, demand, and price of certain commodities, or in how a certain vaccine affects a disease; or in the relationship of rainfall, temperature, and humidity; or in the yields of various varieties of wheat. In other words, we are concerned with the problem of determining the relationship between the internal structure of a system and the observed output. This is the function identification problem.

The function identification problem is to find a function model of an experimental system, in symbolic form, that fits the experimental data points of the system. In this problem, fundamental properties of an experimental system are to be determined from observed behavior of that system. The function model is a mathematical idealization that is used as an approximation to represent the output of the system. It is "good fit" to the given experimental data according to a chosen evaluation criterion.

Here we are concerned with the problem of constructing a computer program system to solve the function identification problems. Researchers in artificial intelligence (machine learning) have investigated the mechanisms of machine discovery and designed some machine learning systems to find empirical laws (function models) from the observations. Most of these methods vary widely by task domains. The purpose of this research is to investigate the feasibility of designing a general-purpose machine function identification system which can automatically build a function model to fit the given experimental data. The new method combines the hierarchical genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm to find a highly fit function model approximating the given data points. This approach is a domain independent method of learning. To this end, a machine function identification system, HGSFI, is implemented.

1.2. SOME RELATED EFFORTS

The goal of machine discovery on empirical laws is to find mathematical relations between numeric variables from observation.

During last decade, many AI discovery systems on empirical laws and discovery were developed. The most important class is BACON-like systems, including BACON (Langley, Bradshaw, & Simon, 1983; Langley, Simon, Bradshaw, & Zytkow, 1987), ABACUS (Falkenhainer & Michalski, 1986), FAHRENHEIT (Zytkow, 1987), and IDS (Nordhausen & Langley, 1990). The BACON-like systems are successful in generating equations for some chemical and physical laws. The major components of BACON-like systems focus on discovering numeric laws from experimental data and use a similar approach -- heuristics. The major difference among the various systems lies in the discovery heuristics that each uses in its search for empirical laws. Here we only discuss the detail of the BACON system to introduce the problem.

BACON is a set of concept-learning programs. These programs solve a variety of single concept learning tasks, including "rediscovering" such classical scientific laws as Ohm's law, Newton's law of universal gravitation, Kepler's law, and Snell's law of refraction. The programs are also capable of using the learned concepts to predict future training instances.

BACON's discovery method consists of a number of interacting techniques. The system begins by gathering data in a systematic fashion, varying one independent term at a time and examining the values of dependent variables. After gathering a set of values, BACON looks for monotonic relations between terms, uses these to define new terms, and recurses until it finds terms with constant values. After finding laws that hold in a given context, the system varies another independent term, using the constants found at the previous level as dependent terms at this higher level of description. This process continues until all terms have been incorporated into some law.

BACON's method for finding constant terms is simple that it can be described here by three straightforward heuristics:

1. If term X has near-constant values, formulate a law involving X.
2. Else, if X increases as Y increases, consider the ratio X/Y and go to step 1.
3. Else, if X increases as Y decreases, consider the product X*Y and go to step 1.

Table 1 presents a simple example of BACON's application of this method in discovering Kepler's third law of planetary motion. This law can be stated as $D^3/P^2 = k$, where D is the distance of a body from its primary, P is the period of that body's revolution around the primary and k is some constant.

Table 1: Discovering Kepler's Third Law of Planetary Motion

Moon	Distance (D)	Period (P)	Term-1 (D/P)	Term-2 (D^2/P)	Term-3 (D^3/P^2)
A	5.67	1.769	3.203	18.153	58.15
B	8.67	3.571	2.427	21.036	51.06
C	14.00	7.155	1.957	27.395	53.61
D	24.67	16.689	1.478	36.459	53.89

The table presents Borelli's original data for Jupiter's satellites, which contain a substantial amount of variation (Langley, Simon, Bradshaw, and Zytkow, 1987). BACON begins by noting that D and P increase together, leading it to consider the ratio D/P. This term is not constant, but its values decrease as those of D increase; this leads BACON to define the product D^2/P. Again, the values of this term are not constant, but its values increase as those of D/P decrease. As a result, the program considers the term D^3/P^2. The values of this term are constant (within the acceptable range of 7.5%), so BACON formulates a law to this effect. The same method can be used to discover a variety of numeric laws.

BACON's main contribution is in the area of quantitative discovery relating real-value variables, the use of rule-space operators to create new (product, quotient, slope, and intercept) terms as combinations of existing terms, and its ability to recast the training instances on the basis of developed hypotheses. There are some difficulties with the current BACON programs, however. BACON is unable to handle noisy training instances. The triggering of the constancy detectors, for instance, is based on the near equality of the values seen in as few as two training instances. Such calculations are highly sensitive to noise. The slope detectors are similarly sensitive.

BACON can handle only relatively simple concept formation tasks of function identification. The approach will result in a huge search space if BACON increases its rule spaces to solve general function identification tasks.

The other BACON-like systems have similar problem -- combinatorial explosion.

1.3. DISCUSSION

Traditional artificial intelligence learning systems for function identification problems are typically based on heuristic rule sets which guide the search process. Most function identification problems, however, are not readily approached by standard rule-based search techniques. For instance, it is difficult to apply heuristic rules to general function identification problems. Not only are the combinations of rules that govern the movement of learning systems very large, but the rules themselves need to change. Rules that might work in one realm need adjustment to work in another realm. The characteristically large search spaces pose formidable obstacles for traditional search techniques. The combinatorial complexity of such problems is a major deterrent to the application of most simple solution strategies.

One technique for solving these kinds of problems, called genetic algorithms, comes from analogy to nature. These algorithms are an outgrowth of a theory of adaptation developed by John Holland (1975). They are motivated by standard models of heredity and evolution in the field of population genetics, embodying abstractions of the mechanisms of adaptation present in natural systems. Nature provides the best demonstration of the power of genetic search wherein the best suited "structures" of organisms evolve in response to environmental pressures. This genetic metaphor encompasses a wide range of (domain independent) search strategies. It is a very flexible way to get computers to learn how to solve problems for themselves. Smith used genetic algorithms to build his general model machine learning system to solve a simple maze walk problem and the problem of making bet decisions in draw poker (Smith 1980). Koza applied hierarchical genetic algorithms to breed populations of computer programs to solve a wide range of problems such as simple robotic planning, sequence induction, symbolic function identification, automatic programming, and so on (Koza 1989, 1990). The author combines Koza's hierarchical genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm to build the general-purpose machine function identification system which can find a "good fit" function model to a given sample data set.

2. HGSFI -- A MACHINE FUNCTION IDENTIFICATION SYSTEM

2.1. INTRODUCTION

A learning system is a system that improves its performance with respect to a given task domain over time through its interactions with the task environment (Smith 1980). The mechanisms by which such a system manipulates its knowledge about the task environment in response to these interactions constitute the system's methods of learning. In constructing an artificial learning system, the particular methods employed determine, to a large extent, the ultimate generality of the system. A learning system capable of functioning in a variety of task domains necessarily requires the presence of domain independent methods of learning. **HGSFI** (Hierarchical Genetic System for symbolic Function Identification) is a general machine learning system for function identification. It manipulates a population of individual function models to find a good fit function model for the given experimental data points. The learning component of the HGSFI takes advantage of two kinds of domain independent techniques -- genetic algorithms and nonlinear regression algorithms -- to attempt to identify a highly fit function model and to optimize its coefficients for the given data set. Section 2.2 introduces the background knowledge the HGSFI uses. Section 2.3 presents the knowledge representation of the HGSFI. Section 2.4

presents the fitness function of the HGSFI which plays important role in this function identification system. The learning paradigm of the HGSFI learning system is presented in section 2.5.

2.2. BACKGROUND

In the machine function identification system, HGSFI, we combine a symbolic computing method and a numeric computing method to identify function models. Section 2.2.1 introduces the background of genetic algorithms. Section 2.2.2 presents the optimization technique of coefficients of function models -- nonlinear regression.

2.2.1. GENETIC ALGORITHMS

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics (Goldberg 1989). John Holland is the primary founder of the field of genetic algorithms. In his pioneer book of genetic algorithms -- **Adaptation in Natural and Artificial Systems** (Holland 1975), Holland showed that an adaptive learning system, called a genetic algorithm, can successfully produce new generations. The new generations can improve their own performance based on their previous performance by evolution. The evolutionary process can be viewed (in a simplified form) as a world where simulated organisms compete to survive. Individuals (artificial organisms) in a genetic algorithm are potential solutions to the problem. The individuals that do well in an environment survive and produce more offspring than the individuals that do poorly. These offspring will inherit some of the characteristics that allowed the parents to survive. Each offspring may also experience some changes through mutation or due to the combination of its parents' characteristics (crossover). If these changes help it to survive, the individual will be able to pass the new characteristics to its offspring. If the changes are detrimental to its survival, the individual dies out. In this way, organisms that are better suited to a particular environment are produced (Ayala and Valentine 1979). This feature of natural selection -- the ability of a population of organisms to explore its search space in parallel and combine the best findings through crossover and mutation -- is exploited when genetic algorithms are used. In a similar manner, genetic algorithms combine survival of the fittest among artificial organisms with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. For example, in function identification problems, the artificial organisms are function models which can be represented as hierarchical tree structures. In every generation, a new set of artificial organisms is created using sub-structures and pieces of the more fit from the old generation; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search parts with expected improved performance (Goldberg 1989).

2.2.1.1. BASIC GENETIC ALGORITHM

A genetic algorithm to solve a problem basically has 5 components (Davis 1987):

1. a chromosomal representation of solutions to the problem,
2. a way to create an initial population of solutions,
3. an evaluation function that plays the role of the environment, rating solutions in terms of their "fitness",
4. genetic operations such as crossover and mutation, that alter the composition of children during reproduction, and
5. values for the parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc.)

Using a genetic algorithm, one represents strategies as chromosomes. Each chromosome serves a dual purpose: it provides a representation of the problem solution, and it also provides the actual material which can be transformed to yield new genetic material for the next generation. In most applications of genetic algorithms, chromosomes (or population members) are represented as bit strings -- lists of 0's and 1's. For example, the seven-bit string A = 0111000 may be represented symbolically as follows:

$$A = a_1a_2a_3a_4a_5a_6a_7$$

Here each of the a_i represents a single binary feature or detector (in accordance with natural analogy, it is called the a_i's genes), where each feature may take on a value 1 or 0. Meaningful genetic search requires a population of strings. Bit strings have been shown to be capable of usefully encoding a wide variety of information, and they have been shown to be effective representation mechanisms in unexpected domains (function optimization and classifier systems, for example). The properties of string-based representations for genetic algorithms have been extensively studied, and a good deal is known about the genetic operators and parameter values that work well with them (Grefenstette 1987; Schaffer 1989; Davis 1987; Wright 1991).

Examples of other representations include real-vectors (Wright 1991), variable-element lists (Grefenstette 1987), and hierarchical structure representation (see next section).

Genetic algorithms begin with an initial population.

Initialization routines vary. For some purposes, a good deal can be learned by initializing a population randomly. Moving from a randomly-created population to a well adapted population is a good test for the algorithm, since the critical features of the final solution will have been produced by the search and recombination mechanisms of the algorithms, rather than the initialization procedures. For some applications, it may be expedient to initialize with more directed methods such as weighted random initialization and initialization by perturbing the results of a human solution to the problem.

There are a great many properties of evaluation functions that enhance or hinder a genetic algorithm's performance. The evaluation function plays the role of the environment, rating solutions in terms of their fitness. In natural populations fitness is determined by an organism's ability to survive predators, pestilence, and the other obstacles to adulthood and subsequent reproduction. In function optimization applications, the fitness function is a rescaling of the objective function **f** of the given problem of genetic algorithms. Intuitively, we can think of the fitness function as some measure of profit, utility, or goodness that we want to maximize. Copying individuals according to their fitness values means that individuals with a higher value have a higher probability of contributing one or more offspring in the next generation.

The genetic operators are defined with respect to the "genetic" representation of a structure. The genetic operators manipulate structures in a population independently of any interpretation (i.e. without regard to the specific task domain), producing new structures for testing (via interpretation). The three primary operations for modifying the structures undergoing adaptation are Darwinian fitness proportionate reproduction, crossover (recombination) and mutation.

Reproduction is a process in which population members are probabilistically copied in such a way that the more fit population members are more likely to be selected to contribute one or more offspring in the next generation. This operation is an artificial version of natural selection. Once a population member has been selected for reproduction, an exact replica of the member is made. The population member is then entered into a mating pool, a tentative new population, for further genetic operation action.

In crossover, the attributes of two population members are combined to produce two offspring. After reproduction, the crossover operation may proceed in two steps. First, members of the newly reproduced population are randomly mated. Second, one point in each parent is selected independently and randomly according to a probability distribution. In the binary string representation, for instance, the crossover operator recombines two strings from the population by exchanging string segments. Two strings are selected from the population for "mating". A crossover point in the strings is chosen randomly. For fixed length string representations, there are L - 1 possible crossover points (each equally likely of being chosen) if the strings are of length L. The strings are then "broken" at the crossover point and recombined so that each new string consists of the initial segment of one of the original strings and the terminal segment of the other. The following example illustrates the one point crossover operation:

```
Parent A:  0 1 1 1 0 0 0 1 0
Parent B:  1 0 0 1 1 0 0 1 1
                  |
Parent A:  0 1 1 1|0 0 0 1 0
                  |
Parent B:  1 0 0 1|1 0 0 1 1
                  |
Child  1:  0 1 1 1|1 0 0 1 1
                  |
Child  2:  1 0 0 1|0 0 0 1 0
                  |
```

Figure 1: single point crossover

The crossover points are marked with separator symbol "|". The left part of child 1 from parent A, and the right comes from parent B. The left part of child 2 comes from parent B, and the right comes from parent A.

In mutation, some of the attributes of a single population member are changed to produce a single offspring. The mutation operator generates a new string by modifying the values of one or more positions in an existing string. An individual member (string) is selected from the population as before. The position(s) in the string to be modified are determined by a random process where by each position has a small probability of being chosen, independently of what happens at other positions. For each string position chosen, a new value is selected randomly from the set of possible values for that position.

The fifth major component is values for the parameters used by genetic algorithms. These parameters, for example, include population size, crossover rate, mutation rate, number of generations to be run, and so on. The selection of the population size is the most important choice. The population size must be chosen with the complexity of the problem in mind. Effective values of the parameters used in the running of genetic algorithms have been studied intensively for string-based representation, and less intensively for other types of representations. Each combination of genetic operators, representation, and the problem has its own characteristics.

2.2.1.2. HIERARCHICAL GENETIC ALGORITHM

One powerful representation scheme for chromosomes (or population members) is the hierarchical structure representation developed by John R. Koza (Koza 1989 and 1990). In a hierarchical structure representation, population members for a particular domain of interest are represented by hierarchical trees. For instance, the cubic polynomial model

$$y = c_1 + c_2x + c_3x^2 + c_4x^3$$

can be represented as the following hierarchical tree.

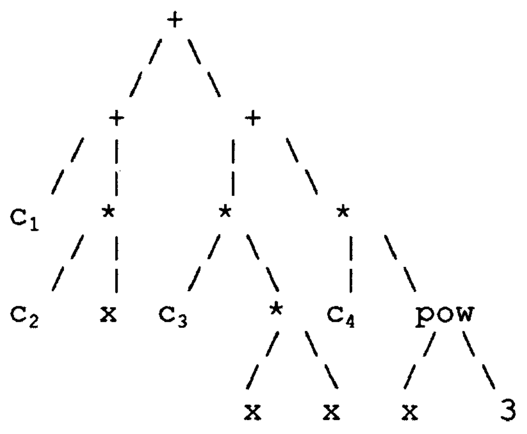


Figure 2: One hierarchical tree for the cubic polynomial model

Note that this representation is not unique for the function model. Another hierarchical tree to represent the cubic polynomial model is shown in Figure 3.

Suppose that in a function identification problem the available set of n functions is **F**={f_1, f_2, ..., f_n} and the available set of m terminals is **T**={a_1, a_2, ..., a_m}. The "terminals" may be variable atomic arguments or constants. Each particular function f in **F** takes a specified number z(f) of arguments b_1, b_2, ..., $b_{z(f)}$. Depending on the particular problem of interest, the functions may be standard arithmetic operations (such as addition, subtraction, multiplication, and division), standard mathematical functions (such as exp, sin, etc.).

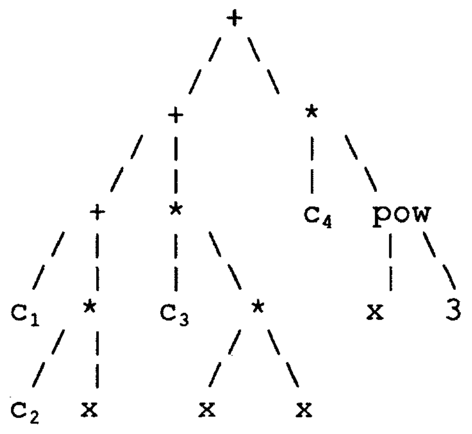


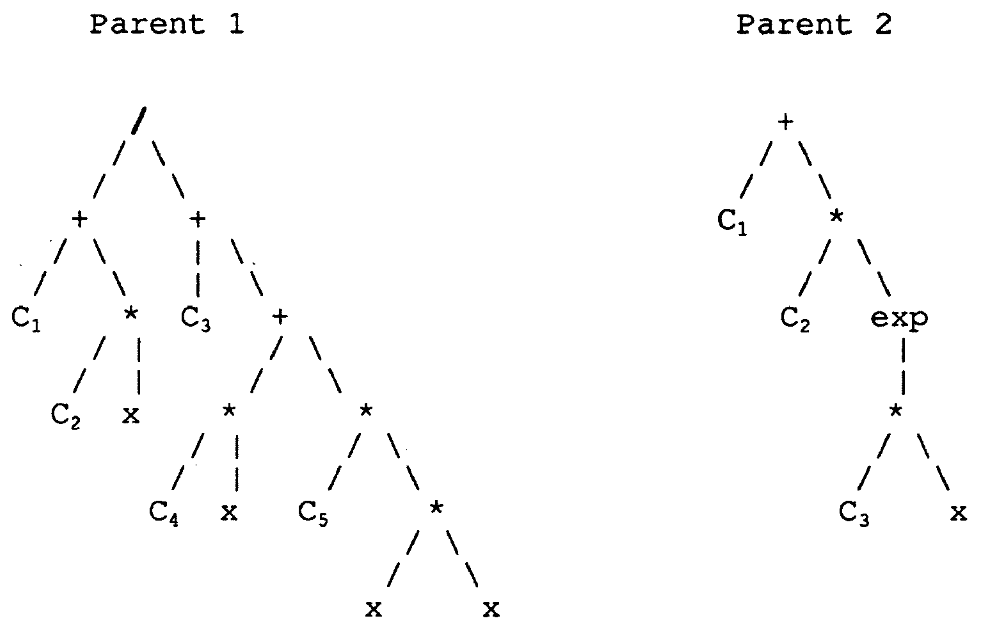
Figure 3: Another hierarchical tree for the cubic polynomial model

Generation of the initial random population begins by selecting one of the functions from the set **F** at random to be the root of the tree. Whenever a point is labeled with a function that takes k arguments, then k lines are created to radiate out from the point. Then, for each line so created, an element is selected at random from entire combined set **S** = **F** U **T** (which is the set of functions and terminals) to be the label for the endpoint of that line. If a terminal is chosen to be any point, the process is then complete for that portion of the tree. If a function is chosen to be the label for any such point, the process continues.

The probability distribution over the terminals and functions in the combined set **S** and the number of arguments taken by each function implicitly determines an average size for the trees generated by this initial random generation process. In genetic algorithms, this distribution is usually a uniform random probability distribution over the entire set **S** (with the exception that the root of the tree must be a function).

Crossover can be implemented as the following. First, two parents are chosen from the current population with a probability proportional to their fitness. Then one point in each parent is selected randomly and independently according to a probability distribution. The "crossover fragment" for a particular parent is the rooted sub-tree whose root is the crossover point for that parent and where the sub-tree consists of the entire sub-tree lying below the crossover point.

The first offspring is produced by deleting the crossover fragment of the first parent from the first parent and then impregnating the crossover fragment of the second parent at the crossover point of the first parent. The second offspring is produced in a symmetric manner.



Parent 1: $\dfrac{C_1+C_2x}{C_3+C_4x+C_5x^2}$; *Parent* 2: $C_1+C_2e^{C_3x}$

Figure 4: The two parental hierarchical trees

For example, consider the two parental hierarchical trees in Figure 4. Assume that the points of trees are numbered in a depth-first way starting at the left. Suppose that the seventh point (out of the 17 points of the first parent) was selected as the crossover point for the first parent and that the fifth point (out of the 8 points of the second parent) was selected as the crossover point of the second parent. The two crossover fragments (sub-trees) are shown in Figure 5.

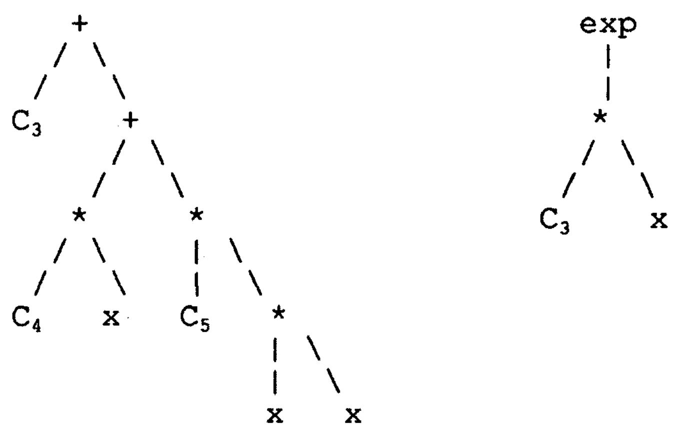


Figure 5: The two crossover fragments

The two offspring resulting from crossover are shown in Figure 6. The result function models are

Offspring 1: $\dfrac{C_1+C_2x}{e^{C_3x}}$

Offspring 2: $C_1+C_2*(C_3+C_4x+C_5x^2)$

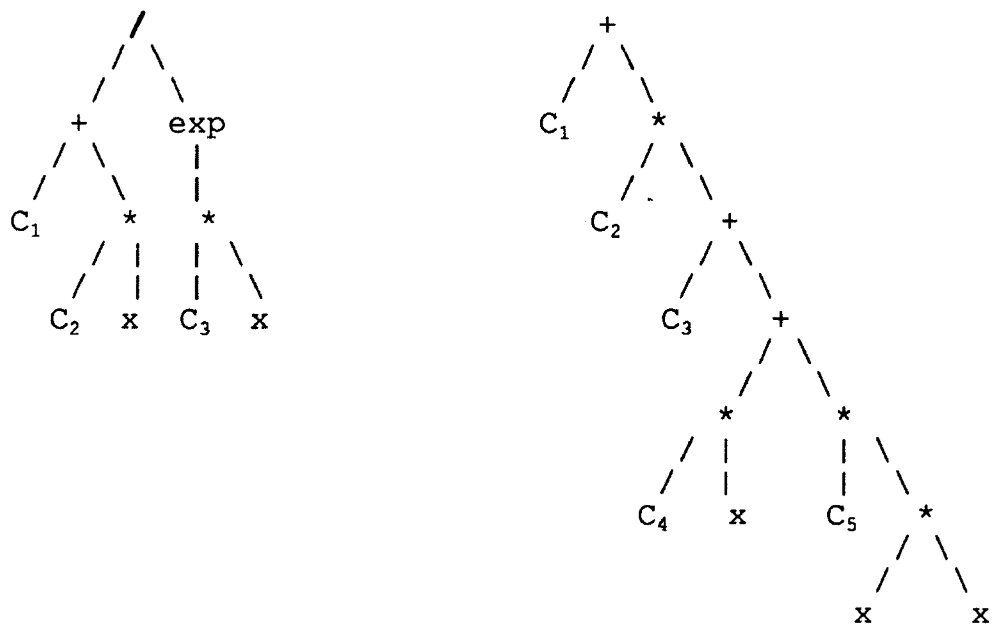


Figure 6: The two offspring resulting from crossover

The mutation operation selects a point of a hierarchical tree at random. This operation removes whatever is currently at the selected point and inserts a randomly generated sub-tree at the selected point of a given tree.

Koza used the LISP programming language to implement his hierarchical genetic system. Individuals in his system are represented as LISP S-expressions. For instance, the function model

$$C_1+C_2e^{C_3x}$$

can be represented as the following LISP S-expression:

$$(+\ C_1\ (*\ C_2\ (\exp\ (*\ C_3\ x))))$$

which is graphically depicted in figure 4. He defined the "raw fitness" of any LISP S-expression as the sum of the distances (taken over all the environmental cases) between the points in the range space returned by the S-expression for a given set of arguments and the correct points in the range space. If the S-expression is integer-valued or real-valued, the sum of distances is the sum of absolute values of the differences between the numbers involved. In particular, the raw fitness r(i,t) of an individual LISP S-expression i in the population of size P_s at any generational time step t is

$$r(i,t) = \sum_{j=1}^{N_e} | S(i,j) - C(j) |$$

where S(i,j) is the value returned by S-expression i for environmental case j (of N_e environmental cases) and C(j) is the correct value for environmental case j. Then the adjusted fitness r(i,t) is defined as

$$a(i,t) = \frac{1}{1+r(i,t)}.$$

The adjusted fitness is larger for better individuals in the population. It lies between 0 and 1. Finally, he defined the evaluation function in his system as the following normalized fitness function:

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{P_s} a(k,t)}$$

The normalized fitness not only ranges between 0 and 1 and is larger for better individuals in the population, but the sum of the normalized fitness values is 1.

Using hierarchical structure representation of population members, we can easily represent the complex structures whose size and shape are dynamically determined (rather than predetermined in advance), and handle the operators of recursions, iteration, and compositions of functions. Koza applied hierarchical genetic algorithms to sequence induction, automatic programming, planning, and function identification (Koza 1990).

2.2.2. NONLINEAR REGRESSION

Regression analysis is a statistical technique for investigating and modeling the relationship between variables. One of the common situations in regression analysis is that of data which consist of observed, univariate responses y_k known to be dependent on corresponding inputs x_k. This situation may be represented by the regression equations

$$y_k = f(x_k, B) + \varepsilon_k \tag{2.1}$$

where f(x, B) is the known response function, x is an independent variable, $B = [b_1, b_2, \ldots, b_M]^T$ is an M-dimensional vector of parameters to be estimated, the ε_k represent a random error from a distribution with mean zero and unknown variance σ^2, and the subscript $k = 1, 2, \ldots, N$ ranges over the N observations. The sequence of values of the independent variable $\{x_k\}$ is treated as a fixed known sequence of constants, not random variable (Gallant 1987).

A linear regression model is the model in which all the parameters appear linearly. That is, the response variable y has some linear relationship with the unknown parameters b_1, b_2, ..., b_M. For instance, the following two regression models are linear in the parameters.

$$y=b_1+b_2x+b_3x^2+b_4x^3 \qquad \textit{where } B=[b_1\ b_2\ b_3\ b_4]^T$$

$$y=b_1+b_2x+b_3e^{x} \qquad \textit{where } B=[b_1\ b_2\ b_3]^T$$

A nonlinear regression model is one in which at least one of its parameters appears nonlinearly. Frequently, nonlinear regression models arise in instances where a specific scientific discipline specifies the form that data ought to follow, and this form is nonlinear in the parameters. For instance, the logistic model is

$$y = \frac{\alpha}{1.0 + \gamma e^{\beta x}}$$

which produces sigmoidal or "S-shaped" growth curves. This model is widely used in biology, agriculture engineering, and economics. In this case $B = [\alpha\ \beta\ \gamma]^T$. Another example is a set of responses that is known to be periodic in time but with an unknown period function for such data is

$$f(x,B) = b_1+b_2\cos\beta_1x+b_3\sin\beta_2x.$$

where $B = [b_1\ b_2\ \beta_1\ b_3\ \beta_2]^T$.

A univariate nonlinear regression model can be written in a convenient vector form

$$Y = F(B) + E \qquad (2.2)$$

where

$$Y = [y_1\ y_2\ \dots\ y_N]^T,$$

$$F(B) = [f(x_1, B)\ f(x_2, B)\ \ldots\ f(x_N, B)]^T,$$

$$E = [\varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_N]^T.$$

The error sum of squares for the nonlinear model and the given data is defined as

$$S(B) = \sum_{k=1}^{N} [y_k - f(x_k, B)]^2 \qquad (2.3)$$

Using vector form, equation (2.3) becomes

$$S(B) = [Y-F(B)]^T[Y-F(B)] = |H(B)|^2 \qquad (2.4)$$

where

$$\begin{aligned} H(B) &= Y - F(B) \\ &= [\{y_1-f(x_1,B)\}\ \{y_2-f(x_2,B)\}\ \ldots\ \{y_N-f(x_N,B)\}]^T \\ &= [h_1(B)\ h_2(B)\ \ldots\ h_N(B)]^T \end{aligned}$$

Note that since y_k and x_k are fixed observations, the sum of squared residuals is a function of B. We denote by B_{min}, a least squares estimate of B, that is a value of B which minimizes S(B). To find the least squares estimate B_{min} we need to differentiate equation (2.3) with respect to B. This provides the M normal equations, which must be solved for B_{min}. The normal equations take the form

$$\sum_{k=1}^{N} \{y_k - f(x_k, B_{min})\}\left[\frac{\partial f(x_k, B)}{\partial b_i}\right]_{B=B_{min}} = 0 \qquad (2.5)$$

for i = 1, 2, ..., M, where the quantity denoted by brackets is the derivative of $f(x_k, B)$ with respect to b_i with all B's replaced by the corresponding B_{min}'s which have the same subscript.

Consider a Taylor expansion of S(B) about the point B_0. This takes the form

$$S(B)=S(B_0)+\sum_i \frac{\partial S}{\partial b_i}\Big|_{B=B_0}(b_i-b_{i0})+\frac{1}{2}\sum_{i,j}\frac{\partial^2 S}{\partial b_i \partial b_j}\Big|_{B=B_0}(b_i-b_{i0})\,(b_j-b_{j0})+\ldots$$

It can approximately be expressed as

$$S(B) \doteq S(B_0) + (B-B_0)^T \cdot G(B_0) + \frac{1}{2}(B-B_0)^T \cdot D(B_0)\cdot(B-B_0) \qquad (2.6)$$

where $G(B_0)$ is the gradient vector of function S(B) at the point B_0, i.e.

$$G(B_0) = \nabla S(B)\big|_{B=B_0} = J^T(B_0)H(B_0) \qquad (2.7)$$

The matrix J is the N*M Jacobian matrix of the vector H(B).

$$J(B_0) = \begin{bmatrix} \frac{\partial h_1(B)}{\partial b_1} & \cdots & \frac{\partial h_1(B)}{\partial b_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_N(B)}{\partial b_1} & \cdots & \frac{\partial h_N(B)}{\partial b_M} \end{bmatrix}_{B=B_0}$$

$D(B_0)$ is the M*M Hessian matrix of function S(B) at the point B_0 whose components are the second partial derivatives, i.e.

$$[D]_{ij}=\frac{\partial^2 S(B)}{\partial b_i \partial b_j}\Big|_{B=B_0} = \sum_{k=1}^{N}[h_k(B)\frac{\partial^2 h_k(B)}{\partial b_i \partial b_j}+\frac{\partial h_k(B)}{\partial b_i}\cdot\frac{\partial h_k(B)}{\partial b_j}]_{B=B_0}$$

Thus D(B) is readily seen to have the form

$$D(B) = J^T(B)J(B) + \sum_{k=1}^{N} h_k(B)H_k(B) \qquad (2.8)$$

where $H_k(B)$ is the Hessian matrix of $h_k(B)$.

In the approximation of (2.6), the gradient of S(B) around point B_0 is easily calculated as

$$\nabla S(B) = G(B_0) + D(B_0)\cdot(B - B_0) \qquad (2.9)$$

The minimum point B_{min} satisfies

$$D(B_0)\cdot(B_{min} - B_0) + G(B_0) = 0 \qquad (2.10)$$

If D is positive definite, this suggests the general iterative scheme

$$B_{i+1} = B_i - D^{-1}G(B_i) = B_i - D^{-1}J^T(B_i)H(B_i) \qquad (2.11)$$

where D is the Hessian matrix of function S(B) at the point B_i.

Equation (2.11) is called as Newton's iterative method. This method requires use of both the gradient vector and the Hessian matrix in computations, hence it places more burden on the user to supply derivatives. Another problem with Newton's method is that the Hessian matrix may not be positive definite at each iteration. Thus the method requires modification to insure that the resultant method is acceptable but still retains the desirable characteristics of Newton's method.

Let us consider the Gauss-Newton method. This method consists of linear approximation of function H(B). If we expand H(B) using only the first two terms in the Taylor series about B_0, we have

$$H(B) \doteq H(B_0) + J(B_0)(B-B_0)$$

Then

$$G(B) = J^T(B)H(B) \doteq J^T(B_0)H(B_0) + J^T(B_0)J(B_0)(B-B_0)$$

At $B=B_{min}$, $G(B_{min}) = 0$. That is

$$J^T(B_0)J(B_0)(B_{min}-B_0) = -J^T(B_0)H(B_0) \qquad (2.12)$$

So we can get generally iteration equation (2.13).

$$B_{i+1} = B_i - [J^T(B_i)J(B_i)]^{-1}J^T(B_i)H(B_i) \qquad (2.13)$$

Notice that this method is essentially Newton's method using an approximation to the Hessian matrix (2.8) which consists of only the first term $J^T(B)J(B)$ in that expression. Also note that each iteration in the Gauss-Newton method requires that the system of linear equations (2.12) be solved. These equations are in the form of normal equations, and in this case it is usually better to compute

$$J(B_i)(B_{i+1} - B_i) = -H(B_i)$$

to preclude introducing excessive amounts of errors in the components of the solution $(B_{i+1} - B_i)$ (Kennedy and Gentle 1980). If there was every assurance that the residuals $h_k(B)$

were all small near the minimum, we would fell confident that this method would operate like Newton's method due to the form of equation (2.8). Unfortunately, large residual problems occur all too frequently and the Gauss-Newton method often does not behave as Newton's method.

Another method is the steepest descent method. The steepest descent method involves concentration on the sum of squares function, S(B) as defined by equation (2.3) and use of an iterative process to find the minimum of this function. The basic idea is to move, from an initial point B_0, along the vector with the negative vector of the gradient vector of function S(B). For this method, the iteration equation becomes

$$B_{i+1} = B_i - \alpha_i G(B_i) = B_i - \alpha_i J^T(B_i) H(B_i) \qquad (2.14)$$

where α_i is a positive constant. The steepest descent method is seldom used today because it is often slow to converge.

A more frequently used method of computing nonlinear least squares estimators is the Levenberg-Marquardt algorithm (Draper & Smith 1981; Gallant 1987). When J(B) is rank-deficient, or nearly so, in the Gauss-Newton iteration, the problem of computing the $(B_{i+1} - B_i)$ in equation (2.13) is difficult and may admit multiple solutions (Kennedy & Gentle 1980). The Levenberg-Marquardt modification transforms J(B) to a better-conditioned full rank matrix.

The usual statement of the modification is to find (B_{i+1} - B_i) according to

$$[J^T(B_i)J(B_i) + \lambda^2 I](B_{i+1}-B_i) = -J^T(B_i)H(B_i) \qquad (2.15)$$

where λ^2 is a chosen scaler. When λ is very large, the matrix

$$[J^T(B_i)J(B_i) + \lambda^2 I]$$

is forced into being diagonally dominant, so equation (2.15) goes over to be identical to (2.14). On the other hand, as λ approaches zero, equation (2.15) goes over to (2.12).

The Levenberg-Marquardt algorithm is as follows (Press, Flannery, Teukolsky, and Vetterling 1988):

```
(1) Give an initial guess for the set of fitted
    parameters $B_0 = [b_1^{(0)}, b_2^{(0)}, ..., b_M^{(0)}]^T$;
(2) Compute the sum of squared residuals $S(B_0)$ using the
    equation (2.3);
(3) Pick a modest value for $\lambda$, say $\lambda = 0.001$, Set
    iterative count K and J to be 0;
(4) While (J < 2) Do
    (a)  Solve the linear equation (2.15) for $dB_K$ and
         evaluate $S(B_K + dB_K)$ where $dB_K = B_{K+1} - B_K$;
    (b)  If $S(B_k + dB_k) < S(B_k)$ and
            $|S(B_k + dB_k) - S(B_k)| < 0.001$
            J <-- J + 1
         Else J <-- 0;
    (c)  If $S(B_K + dB_K) \geq S(B_K)$, increase $\lambda$ by a factor of
         10 (or any other substantial factor) and go back
         to (4);
    (d)  If $S(B_K + dB_K) < S(B_K)$, decrease $\lambda$ by a factor of
         10, update the trial solution $B_{K+1}$ <-- $B_K + dB_K$, and
         go back to (4);
(5) End
```

Figure 7: The Levenberg-Marquardt algorithm

The Levenberg-Marquardt algorithm appears to enlarge considerably the number of practical problems that can be tackled by nonlinear estimation. Marquardt's method represents a compromise between the linearization (Gauss-Newton) method and the steepest descent method and appears to combine the best features of both while avoiding their most serious limitations. It is good in that it almost always converges and does not "slow down" as the steepest descent method often does (Draper & Smith 1981).

2.3. THE KNOWLEDGE REPRESENTATION OF HGSFI

Knowledge representation is a key issue in any learning system because the representation scheme can severely limit the window by which the system observes its world. In the machine function identification system, HGSFI, the knowledge structures are various function models in the search space. Suppose that the available set of n function operations is $\mathbf{F} = \{f_1, f_2, \ldots, f_n\}$ and the available set of m terminals is $\mathbf{T} = \{x, c_1, c_2, \ldots, c_{m-1}\}$. The search space for the HGSFI is the valid function models that can be recursively created by compositions of the available function operations and the available terminals for the problem. This search space can, equivalently, be viewed as rooted node-labeled trees with ordered branches having internal nodes labeled with the available function operations and external nodes (leaves) labeled with the available terminals. Note that the set of functions and terminals being used in a particular problem should be selected so as to be capable of solving the problem.

For different problems, the function models to fit them are various. Even if the sample data sets are same, there can be many different function models which are "good fit" to the experimental data according to the chosen evaluation criterion. That is, the function models of different size, shape, and complexity need to be tested during the process of function identification. String-based representation schemes do not provide the hierarchical structure and any convenient way to process the dynamically varying and complex structures of function models and to calculate the values of function models. Using the hierarchical structure representation scheme, however, we can represent any complex function model, can easily implement the genetic operations (such as crossover and mutation) as described in section 2.2.1.2, and can efficiently evaluate the values of function models. So the knowledge (i.e. function models) in the HGSFI are represented as hierarchical structures.

2.4. THE FITNESS FUNCTION

In genetic algorithms, each individual in a population is assigned a fitness value as a result of its interaction with the environment. Fitness is the driving force of Darwinian natural selection and, likewise, of genetic algorithms (Koza 1990). The fitness function in genetic algorithms plays the role of the environment. Based on the proportional to the fitness of individuals in a population, genetic algorithms probabilistically select individuals on which to apply the genetic operations of reproduction, crossover, and mutation. (In the most commonly used selection procedure, the probability of selection is proportional to the normalized fitness of the individual.) The fitness of individuals become very important because they can determine whether the individuals survive in the next generation. Proper selection of the fitness function is the key for success of the genetic algorithm process. Choice of the fitness function should base on the problem and coincide with the choice of representation.

Statistians usually use the following two methods to describe the difference between the predicted model f(x, B) and real value y_k (equation 2.1). The first method is the sum of the squares between the given values of observations and computed values of the dependent variable. That is

$$SSResid = \sum_{k=1}^{N} [y_k - f(x_k, B)]^2 \tag{2.16}$$

The another one is the sum of absolute residuals

$$SAResid = \sum_{k=1}^{N} | y_k - f(x_k, B) | \tag{2.17}$$

If we choose the sum of squared residuals to be the performance measure, then we have a performance measure function which is consistent with that of nonlinear regression algorithms. However, if the residual $y_k - f(x_k, B)$ of some point is less than 0.1, then the squared residual will be less than 0.01. And if the residual of some point is greater than 100, then the squared residual of that point will be much greater than 100. That is, the sum of squared residuals is not directly proportional to the residual of each point.

The sum of absolute residuals has linear relationship with the residual of each point. Our experimental results indicate that using the sum of absolute residuals as the performance measure is better than using the sum of squared residuals. In our system, the performance measure Perf(i, t) of an individual function model i in the population at generation t is defined as follows:

$$Perf(i,t) = \sum_{k=1}^{N} | y_k - f_i^{(t)}(x_k, B_i^{(t)}) | \tag{2.18}$$

where $f_i^{(t)}(x_k, B_i^{(t)})$ is the function equation of individual i at generation t. $B_i^{(t)}$ is the estimated parameter vector of individual i at generation t. Note that this definition of Perf(i,t) is same as the raw fitness in Koza's system. The smaller the performance, the better an individual in the population. We might try to directly use this performance to calculate the fitness as follow:

$$f(i,t) = \frac{Perf(i,t)}{\sum_{k=1}^{P_S} Perf(k,t)}$$

where P_s is the population size. However, the performance of the worst individual in a population might be much larger than the performances of other individuals in the population. Then

$$\sum_{k=1}^{P_S} Perf(k,t) \doteq \textit{the performance of the worst guy}$$

In this case, the fitness of the worst individual would be very close to 1, and the fitness of the others would be very close to 0. This fitness function would do a poor job of distinguishing the relative performances of the better individuals in the population. In other words, this performance measure does not stress good performance, the genetic algorithm may fail to converge on good results in a reasonable time and will be more likely to lose the best members of its population. For this reason, we need employ a normalization technique for performance measure. In a manner similar to Koza's system, we define the normalized performance of individual i at generation t as following

$$Perf_{Norm}(i,t) = \frac{1}{1+Perf(i,t)} \qquad (2.19)$$

Then, the fitness of individual i at generation t is calculated as

$$f(i,t) = \frac{Perf_{Norm}(i,t)}{\sum_{k=1}^{P_S} Perf_{Norm}(k,t)} \qquad (2.20)$$

The fitness lies between 0 and 1 and is larger for better individuals in the population. The sum of the fitness values in a population is 1. In our system, we use this fitness to be as the probability of individuals' performance in a population. When we say "proportional to fitness" or "proportional to the probabilities of individuals' performances in a population", we are referring to the fitness as defined above.

2.5. THE LEARNING PARADIGM OF HGSFI

Function identification is a complicated and iterative learning process. Faced with so many indeterminancies and uncertainties, our system should find the "best" model to fit the given sample data set through the iterative process of learning by experiment. Our problem is to so organize matters that we are likely in due learning process to be led to the right conclusion even though our initial choice of the function models may not all be good. Our strategy must be such as to allow any poor initial choices to be rectified as we proceed. To meet these requirements and general-purpose learning requirement, the learning component of the HGSFI takes advantage of two kinds of domain independent techniques -- the hierarchical genetic algorithm and the Levenberg-Marquardt algorithm -- to attempt to identify a highly fit function model and optimize the coefficients of the model for the given sample data set. The learning paradigm of the HGSFI machine function identification system is depicted in Figure 8.

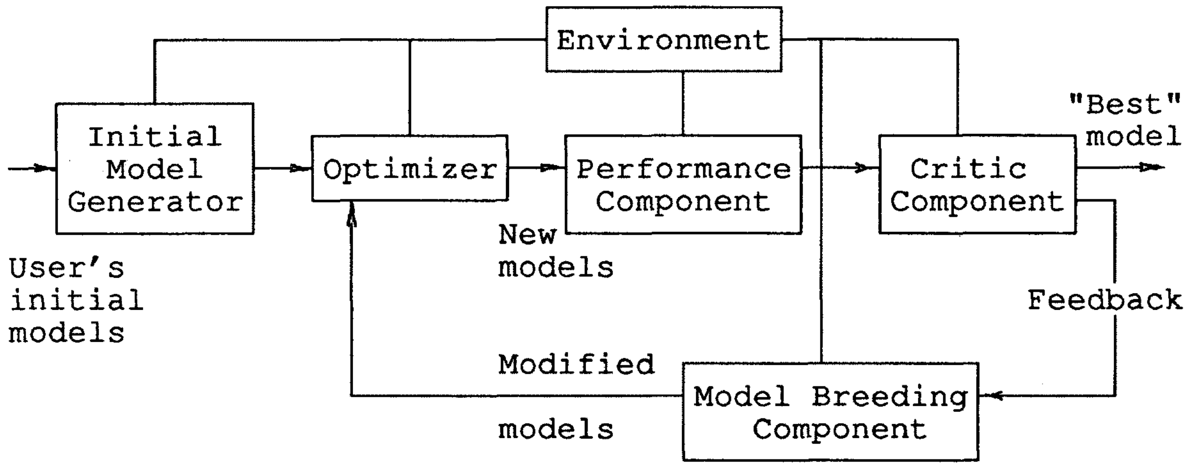


Figure 8: The Learning Paradigm of the HGSFI

This learning system manipulates a population of individual function models to cope with their environment. The environment includes the given sample data set, the possible solution (function model) space, control parameters, and the fitness of individuals in a population. At the beginning, the **initial model generator** randomly generates initial function models (i.e., the original population of function models). The initial individuals may not be good generally. Through the evolution of populations, generation after generation, the initial individual function models and their offspring are gradually modified to adapt to their environment. Note that the "environment" differs for different generational individuals, but is same for the individuals in the same generation.

The **optimizer** (Levenberg-Marquardt algorithm) optimizes the coefficients of input function models and tries to make the function models to best fit the given sample date set. Note that the nonlinear regression subroutines (i.e. optimizer) are embedded in the **evaluate** procedure when the HGSFI is implemented (see section 3.2).

During each cycle through the learning loop (i.e. the evolution process of population of individual function models), the **performance component** (including the **evaluate** procedure) interacts with the environment. It applies the system's current knowledge (i.e. new generation function models) to the sample data set and evaluates performances and fitness of individuals in the new generation. The new generation function models and their fitness are analyzed by the **critic component** (including the **measure** procedure) with respect to the given performance criterion. The critic component compares the fitness of individuals in the population to determine the "best" individual. If some individual satisfies the given performance criterion then the system returns the most fit function model that the system has found, and halts. Otherwise, the system attempts to produce a better population of individual function models through the adaptive changes which consist of the adjustments made by individuals in response to the environmental conditions, and enable the population of individuals to cope with the environment and to continue its existence.

The population of individual function models and their associated fitness are sent as feedback to the **model breeding component** (including the **reproduce, crossover, mutate, permute**, and **elitist** procedures, see section 3.2). Based on the engine of Darwinian reproduction and survival of the fittest, the model breeding component selects individuals to produce new generation of individuals according to their fitness. Individuals that were not selected do not survive. A selected individual will generate a number of offspring in direct proportion to its relative fitness compared with the other individuals. In other words, if the fitness of an individual is twice that of another individual, the first individual will have on average twice the number of offspring in the succeeding generation.

Then some of the selected individuals are recombined with the others of the selected individuals to produce their offspring which inherit the parents' merits and adjust themselves to environmental fluctuations. Some of the selected individuals make some changes by themselves through mutating the "genes" (i.e. the members of set $\mathbf{S} = \mathbf{F} \cup \mathbf{T}$) they carry. These inheritable adjustments made by individuals in response to specific environmental conditions are a facet of the individuals to cope with their environment and to continue survival. Here the genetic algorithm is no simple random walk. It efficiently exploits the wealth of information by genetic operations to speculate on new search points with improved performances.

3. THE IMPLEMENTATION OF HGSFI

The HGSFI is implemented using the C programming language. The main control structure of the system basicly is same as that of the GENESIS (Grefenstette 1984). The HGSFI processes populations of hierarchical structures. It consists of four programs: setup, main, report, and plotmodl.

The program SETUP creates the input parameter file for the MAIN. The user can give values of control parameters or use the default values of control parameters. The control parameters include population size, crossover rate, mutation rate, permutation rate, maximum depth of initial hierarchical structures, maximum depth of hierarchical structures, converged condition, maximum generation the system runs and so on.

The MAIN program implements the learning process which attempts to find a highly fit function model for the given sample data set, using the hierarchical genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm.

The REPORT program generates the statistical report summarizing the mean and variance of a number of performance measures of several runs of the HGSFI, and outputs the "best" function model, its estimated coefficients, and the summary statistics of nonlinear regression for each run of the HGSFI.

The PLOTMODL program plots the observed curve and the predicted curve on the screen.

3.1. DATA STRUCTURES

The MAIN program consists of an implementation of the general-purpose machine function identification system using the hierarchical genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm. The hierarchy chart of data structures used in the MAIN program is shown in Figure 9.

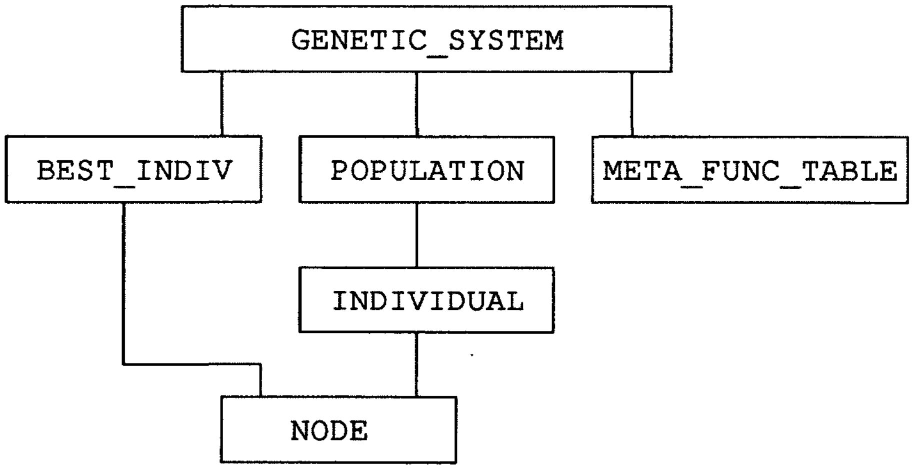


Figure 9: The hierarchy chart of data structures

3.1.1. INDIVIDUAL AND NODE

INDIVIDUAL is the basic class of data structures in the HGSFI. It defines the basic attributes of an individual. The definition of the INDIVIDUAL structure is shown in Figure 10.

```
typedef struct {
   struct NODE *model;       /* hierarchical tree of the
                                function model            */
   int         numOfNodes;   /* number of nodes in the
                                hierarchical tree         */
   double      performance;  /* normalized performance    */
   double      fitness;      /* fitness of the individual */
   double      sumOfSqResid; /* sum of squared residuals  */
   double      *constList;   /* constList[numOfConst]     */
   int         numOfConst;
   unsigned char needEvaluation;
} INDIVIDUAL;
```

Figure 10: The definition of the INDIVIDUAL structure

The attribute **model** of the INDIVIDUAL structure is a hierarchical tree which consists of the base structure type NODE. The **NODE** structure describes the attributes of a node in the hierarchical tree of a function model. The definition of the NODE structure is depicted in Figure 11. Each node contains 5 attributes. The attribute **type** indicates the node is a constant, independent variable, or function operation node.

```
struct NODE {
  char        type;  /* C -- constant node,
                        V -- independent variable node,
                        F -- function operation node    */
  int         index; /* index in FuncTable or constList */
  double      value;
  struct NODE *leftChild;
  struct NODE *rightChild;
};
```

Figure 11: The definition of the NODE structure

The **index** attribute is the index in the function table (see section 3.1.3) for a function node or in the constant list of the model for a constant node. The attribute **value** keeps the value of a constant if the node is a constant node. The other two attributes are **leftChild** and **rightChild** which point to the children nodes of the node.

In the INDIVIDUAL structure, the attribute **numOfNodes** is the number of nodes to constitute the model. In order to easily change the values of constants in a model during computation of nonlinear regression, we set a **constList** attribute for each individual. The constant list is an array to save the values of the constants (i.e. estimated parameters) of a model. The attribute **needEvaluation** indicates whether the performance of the model need to be evaluated in the current generation. If the structure of a model has no changes from preceding generation to the current generation, the performance of the model does not need to be evaluated again.

3.1.2. POPULATION

The population structure maintains some basic information about individuals in the current generation. The attributes of the **POPULATION** structure are shown in Figure 12. **AveCurrentPerf** and **AveCurrFitness** are calculated over the entire population whenever the population is evaluated. **BestCurrFitness** and **WorstCurrFitness** keep the best fitness and the worst fitness in the generation, respectively. In addition, the population structure maintains two variables **BestGuy** and **WorstGuy**, which point to the best member and the worst member in the current generation, respectively.

```
int    PopSize;                /* population size            */
INDIVIDUAL *Old;               /* INDIVIDUAL  Old[PopSize]   */
INDIVIDUAL *New;               /* INDIVIDUAL  New[PopSize]   */
double AveCurrentPerf;         /* average normalized perf. in
                                  current generation         */
double AveCurrFitness;         /* average fitness in current
                                  generation                 */
double BestCurrFitness;        /* best fitness in current
                                  generation                 */
double WorstCurrFitness;       /* worst fitness in current
                                  generation                 */
int    BestGuy;                /* point to best member in
                                  current generation         */
int    WorstGuy;               /* point to worst member in
                                  current generation         */
```

Figure 12: The definition of the POPULATION structure

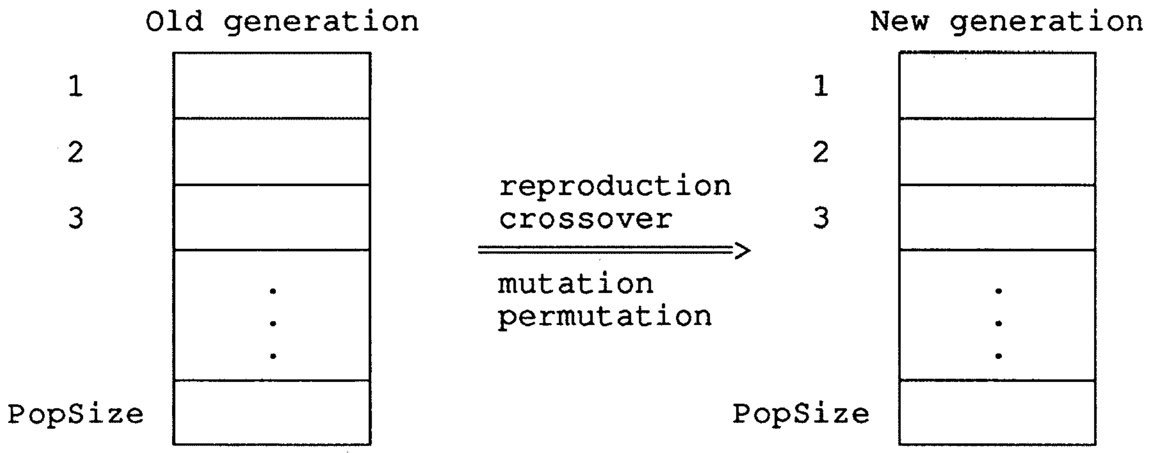


Figure 13: Schematic of nonoverlapping population

In the HGSFI, we apply genetic operators to an entire population at each generation, as shown in Figure 13. To implement this operation cleanly, we utilize two nonoverlapping populations **Old** and **New**, thereby simplifying the birth of offspring and the replacement of parents. Note that the population size can be set by the user.

3.1.3. GENETIC_SYSTEM

GENETIC_SYSTEM is the top class of data structures in the HGSFI. It consists of the following global variables to control the process of the genetic algorithm and to save statistical information.

```
BEST_INDIV *BestSet;  /* BEST_INDIV BestSet[MaxGen];       */
META_FUNC_TABLE FuncTable[MAX_FUNC_SIZE];
double BestPerf;      /* best performance seen so far      */
double BestFitness;   /* best fitness seen so far          */
int    MaxGenerations; /* maximum number of generations    */
int    Generation;    /* generation counter                */
int    InitMaxDepth;  /* maximum depth of initial function
                         models                            */
int    MaxDepth;      /* maximum depth of function models */
double CrossoverRate;
double MutationRate;
double PermutationRate;
int    NumOfPoints;   /* number of sample points           */
double *DataX;        /* double DataX[NumOfPoints];        */
double *DataY;        /* double DataY[NumOfPoints];        */
double CorrectSSTotal;/* total of corrected sum of squares*/
double UncorSSTotal;  /* total of uncorrected sum of
                         squares                           */
double ConvergentValue;/* convergent condition of HGSFI   */
int    ConvergedFlag; /* converged flag of the system      */
          .
          .
          .
char   EliteFlag;     /* use elitist selection strategy    */
char   RegressionFlag;/* use nonlinear regression to
                         optimize the coefficients of
                         function models                   */
char   SimplifyFlag;  /* simplify function model           */
          .
          .
          .
```

Figure 14: The attributes of GENETIC_SYSTEM

In order to keep track of "best" individuals in all generations, the system has a global array, **BestSet**, to save them. The member structure of BestSet is shown in Figure 15.

```
typedef struct {
   struct NODE *model;       /* hierarchical tree of the
                                function model          */
   int         numOfNodes;   /* number of nodes in the
                                hierarchical tree       */
   double      performance;  /* normalized performance  */
   double      fitness;      /* fitness of the individual */
   double      sumOfSqResid; /* sum of squared residuals */
   double      *constList;   /* constList[numOfConst]   */
   int         numOfConst;
   int         gen;          /* generation number       */
   int         trials;       /* trial number            */
} BEST_INDIV;
```

Figure 15: The definition of BEST_INDIV structure

In the HGSFI, there is a global variable, **FuncTable**, to save the set **S** which is the union of terminal set **T** and function operation set **F**. The member structure (**META_FUNC_TABLE**) of FuncTable is shown in Figure 16. The schematic of the function table is depicted in Table 2.

```
typedef struct {
   char   name[10];       /* function name                */
   int    numOfVars;      /* -1 -- constant; 0 -- variable;
                             1 or more -- functions       */
   int    restriction;    /* 0 -- No restriction;
                             1 -- first variable != 0;
                             2 -- second variable != 0;
                             3 -- first variable > 0      */
   double value;          /* only for constants           */
} META_FUNC_TABLE;
```

Figure 16: The definition of META_FUNC_TABLE structure

Table 2: The Schematic of Function Table in the HGSFI

index	name	numOfVars	restriction	value
1	π	0	No	3.1415926
2	random constant	0	No	
3	independent variable	0	No	
4	+	2	No	
5	-	2	No	
6	*	2	No	
7	/	2	second variable != 0	
8	exp	1	No	
9	**log**	1	variable > 0	
...	...	...	...	...

Note that

(1) The initial value of a random constant is randomly generated in the range [-25.0, 25.0] uniformly. After optimization, the value of a random constant can be any floating point number in the range [-MAX_DOUBLE_VALUE, MAX_DOUBLE_VALUE]. Our MAX_DOUBLE_VALUE is defined as 1.7e+140.

(2) The restriction of function operation indicates what kind of restrictions of operands to do the function operation. For example, division operation requires the second operand to not be equal to zero. In this case, if the second operand is zero, then the division function will return

MAX_DOUBLE_VALUE. And **log** is the restricted logarithm function which is defined as

$$\log(x) = \begin{cases} 0, & x < 0 \\ -\infty, & X = 0 \\ \ln(x), & x > 0 \end{cases}$$

where the infinite is MAX_DOUBLE_VALUE.

The minimum set of functions in HGSFI is

F = { +, -, *, / };

the maximum set of functions in HGSFI is

F = {+, -, *, /, minus_sign, **, sq, sqrt, exp, log, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, int}.

The given sample data points $\{x_k, y_k\}$ (k = 1, 2, ..., **NumOfPoints**) are saved in arrays **DataX** and **DataY**, respectively. The total of corrected sum of squared residuals (**CorrectSSTotal**) is defined as

$$CorrectSSTotal = \sum_{k=1}^{N} [y_k - y_{avg}]^2 \qquad (3.1)$$

where N is the number of the sample points, y_{avg} is the average value of y_k. The total of uncorrected sum of squared residuals (**UncorSSTotal**) is defined as

$$UncorSSTotal = \sum_{k=1}^{N} (y_k)^2 \qquad (3.2)$$

The other attributes of GENETIC_SYSTEM are discussed in next two section.

3.2. THE FRAMEWORK OF THE MAIN PROGRAM

The kernel of the main program is the genetic algorithm. The main loop of the main program is an iterative procedure which maintains a constant-size population P(t) of candidate solutions. The main loop is shown in Figure 17.

```
BEGIN
   (1)  Choose the initial population of size $P_s$, call this
        population P(0). Set the generation counter t to be
        0;
   (2)  Evaluate the performance (if needEvaluation for the
        individual) and the fitness of each individual in
        P(t);
   (3)  Measure each individual in P(t) to check
        convergence;
   (4)  If ConvergedFlag is on or
           generation counter t ≥ maximum generation
        then terminate;
   (5)  Increase generation counter by 1;
   (6)  Reproduce P(t) from P(t-1), call population P(t) as
        new_pop, population P(t-1) as old_pop;
   (7)  Crossover P(t);
   (8)  Mutate P(t);
   (9)  Permute P(t);
   (10) If EliteFlag is on then
           Call Elitist;
   (11) For each individual (call i) in P(t) do
         (i)  If new_pop[i].needEvaluation then
              (a) If SimplifyFlag is on then
                     Simplify the new_pop[i].model of
                     individual i at generation t;
              (b) Count the number of nodes in
                  new_pop[i].model;
              (c) Count the number of constants in
                  new_pop[i].model;
              (d) Build constant list of individual i at
                  generation t;
         (ii) Endif;
   (12) Go to (2);
END.
```

Figure 17: The main loop

The user could provide some of initial function models which suit his/her problem. In our experiments, we always chose the initial population P(0) randomly. The method to generate **initial** population P(0) at random was described in section 2.2.1.2. Note that the initial function model trees are controlled by a system parameter, **InitMaxDepth**, which specifies the maximum depth of the randomly generated initial function trees.

The **evaluate** procedure evaluates the performance of each individual in the current generation if the function model of the individual is changed during the process of previous generation (i.e., it needs be evaluated again), and computes the fitness of the individuals in the population. The nonlinear regression subroutines (Levenberg-Marquardt algorithm) are embedded in the evaluate procedure, which optimize the coefficients of the function model of an individual to reduce the sum of squared residuals and to fit the given sample data points. The main loop of the Levenberg-Marquardt algorithm was given in Figure 7 (see section 2.2.2). Note that there are two cutoff values used to stop iterative calculation in our implementation of the LM nonlinear regression algorithm. The first one is the convergence criterion for the sum of squares. When S(B) decreases by less than 0.001 on two consecutive iterations, the fit is considered complete. The second one is the maximum number of iterations allowed. It is controlled by the iterative counter K (see Figure 7). In our experiments described in Chapter 4, it was always set to be 50. In addition, we set the initial value of λ to be 0.001 in our implementation.

The **measure** procedure computes some statistical information such as the average performance and the average fitness of the current generation and calculates performance measures. It compares the fitness of individuals in the current generation to determine the "best" individual and saves the "best" individual into **BestSet**. It measures each individual function model to check if it has converged. This procedure calculates the convergence test value of individual i at generation t as the following

$$Convg(i,\ t) = 1.0 - \frac{\sum_{k=1}^{N} | y_k - f_i^{(t)}(x_k, B) |}{\sum_{k=1}^{N} | y_k |} \qquad (3.3)$$

where $f_i^{(t)}(x, B)$ is the function equation of individual i at generation t. { (x_1, y_1), (x_2, y_2), ..., (x_N, y_N) } is the sample data set of the given problem. If the convergence test value for some individual is greater than the system convergence condition (**ConvergentValue**), then the function model has converged and the global variable **ConvergedFlag** in the system is set as TRUE.

The genetic system uses two techniques for stopping an experiment. These are the maximum number of generations and the convergence condition of the system. The **MaxGenerations** variable is set by the user. It is controlled by an iterator. The system never lets an experiment continue past this ceiling on the number of generations. In each iteration, after evaluating performance of individuals, the system measures each individual function model to check if it has converged. If the system variable **ConvergedFlag** is TRUE, that means some individual function model has satisfied the criterion of the system convergence condition. The **terminate** procedure outputs the statistical information and then stops the process of an experiment.

The **reproduce** procedure implements the process of choosing individuals for the next generation from the individuals in the current generation. The individuals in reproduction pool (**New** population) are chosen from old population by a randomized reproduction procedure that ensures that the expected number of times an individual is chosen is approximately proportional to that individual's fitness.

In the **crossover** procedure, the individuals in reproduction pool are randomly chosen for crossover (employing a user-specified crossover probability,

CrossoverRate). There are two strategies to select a crossover site in our system. One is to choose the crossover point randomly using uniform distribution over the internal and external nodes in the parents. Another one is to use a non-uniform probability distribution over the potential crossover points in the parents that allocates 90% of the crossover points equally amongst the internal (function) nodes of each tree. The remainder of the crossover points are allocated equally amongst the external (terminal) nodes of each tree. Note that if the depth of any offspring tree exceeds the maximum depth of function models (determined by the system parameter **MaxDepth**), then subtrees of nodes whose levels in the offspring tree exceed MaxDepth are removed. For each such subtree, the tree is replaced by a randomly generated terminal (constant or independent variable) node. In our experiments described in chapter 4, we always used the non-uniform strategy to select a crossover site in an individual.

After crossover, some individuals are **mutated** according to a user-specified mutation probability (**MutationRate**). We have two different implementations of the mutation operation in our system.

The first method is similar to Koza's implementation (Koza 1990). In the first implementation, the individuals are randomly selected from reproduction pool for mutation. The mutation operation selects a node of the individual at random. The node can be an internal (function) or external (terminal) node of the tree. This implementation removes whatever is currently at the selected node of a given tree, and inserts a randomly generated sub-tree at the selected mutation node of the individual. To generate a sub-tree, we begin by selecting one of the functions or terminals from set **S** instead of the set **F** (like the generation of initial function models), at random and uniform, to be the root of the sub-tree. In this implementation, approximately MutationRate*PopSize individuals in a population are selected for mutation. The mutation operation is performed on single node of an individual selected for mutation.

Using the second implementation method, approximately MutationRate*(total nodes of all individuals in the current generation) mutations occur per generation. An individual can contain more than one mutation node. The algorithm of the second method is shown in Figure 18.

Our experiments indicated that the second implementation of mutation operation was better for some test problems. The experiments described in chapter 4 were done using the second mutation method.

```
(1)  For i = 1 to MutationRate*(total nodes of all
                   individuals in the current generation)
(2)  Do
     (a) Randomly select an individual from current
         generation, call it as individual j;
     (b) Randomly choose a node of individual j, call this
         node as the mutation node;
     (c) Randomly select a meta function from the function
         table (i.e., set S), call it as new node;
     (d) If new node has same number of variables as the
         mutation node, simply replace the mutation node
         by new node;
     (e) Else if new node is a terminal, replace the sub-
         tree whose root is the mutation node with the new
         node;
     (f) Else if new node is an unary function node, then
         (i)  If the mutation node is a terminal node,
              randomly choose a terminal from set T as the
              operand of new node and use this sub-tree to
              substitute the mutation node;
         (ii) Else remove the right child of the mutation
              node and replace the mutation node with new
              node;
     (g) Else if new node is a binary function node, then
         (i)  If the mutation node is a terminal node,
              randomly choose two terminals from set T as
              the operands of new node and use the sub-tree
              to substitute the mutation node;
         (ii) Else use the left child of the mutation node
              as the first operand of new node, randomly
              select a terminal from set T as the second
              operand of new node, and replace the mutation
              node with this sub-tree;
(3)  Endloop
```

Figure 18: The second mutation method

Note that the mutation operation is controlled by the system parameter **MaxDepth** which specifies the maximum depth for the newly created or modified tree. If the depth of the modified tree exceeds MaxDepth, then subtrees of nodes whose levels in the tree exceed MaxDepth are removed. For each such subtree, the tree is replaced by a randomly generated terminal (constant or independent variable) node.

The **permute** procedure is the process to perform the permutation operation. It operates on only one individual. The individual is randomly selected from new population. The permutation operation selects a function node of the hierarchical tree at random. If the function at the selected node is a binary function, then left child and right child are swapped. Notice that if the function at the selected node is commutative, there is no immediate effect from the permutation operation on the tree.

Two population-maintenance strategies can be employed to direct the evolution. One is a pure selection strategy, in which the individuals are reproduced in proportion to their fitness as above description.

Another one is **elitist** strategy. The elitist policy stipulates that the best individual always survives into the new generation. The elite individual is placed in the last position in new generation, and is not changed through crossover, mutation, or permutation. In the absence of such a strategy (**EliteFlag** is off), it is possible for the best individual to disappear, due to sampling error, crossover, mutation, or permutation.

During the process of genetic operations, if the function model of an individual is changed, then the attribute needEvaluation of the individual is set as TRUE. After processing genetic operations, the models of the modified individuals can be simplified using the set of simplifying rules if the user chooses the simplification option (**SimplifyFlag** is on). The **simplify** procedure simplifies the function equation using the set of simplification rules (i.e., some mathematical rules). For example, the equation

$$C_1 * (C_2 + C_3 * X)$$

can be simplified as

$$C'_1 + C'_2 * X$$

where $C'_1 = C_1 * C_2$ and $C'_2 = C_1 * C_3$.

Using the simplification option can save computer resources: space and time. It may improve overall performance slightly for some problems. However, our experience showed that it could not improve overall performance for most of linear regression model problems and nonlinear regression model problems we tested. So we only applied the simplification to the final stage -- outputing function models to a file or on the screen.

Then the attributes numOfNodes and numOfConst of the modified individuals are re-**count**ed. In addition, the constant lists of the modified individuals are re-built. In the **build**ing constant list procedure, the values of the constants of an individual are adjusted if the individual has some changes after genetic operations. The adjustment of constant values is to increase them in 10 percent or decrease them in 10 percent at random. This adjustment of constants is another form of mutation. It may help the optimizer (Levenberg-Marquardt algorithm) to move from the locally optimal points.

3.3. THE SELECTION OF CONTROL PARAMETER VALUES

Running a genetic algorithm entails setting a number of control parameter values. If poor settings are used, a genetic algorithm's performance can be severely impacted. Finding the optimal control parameter settings can be difficult, because different problems have different optimal values of control parameters. Each combination of genetic operators, representation, and problems has its own characteristic. In this section, we give our empirical selection of control parameters for function identification problems.

(1) The selection of function table (**S**):

In order to identify the function model for the given sample data set, we need first choose the function table which consists of the set of terminals and the set of functions. The function table must, of course, be sufficient to solve the problem. In our experiments, we chose the set of terminals as

T = {random_constant, independent_variable}.

The set **F** of functions was always chosen as

F = { +, -, *, /, **, exp, log, sin, cos }

unless otherwise specified.

(2) Population size (P_S):

The population size affects both the ultimate performance and the efficiency of the system. Genetic algorithms generally do poorly with very small populations, because the population provides an insufficient sample size for most solution spaces. A large population is more likely to contain representatives from a large number of solution space. Hence, genetic algorithms can perform a more informed search. For the function identification problem, relatively large population size is needed. As a result, a large population discourages premature convergence to suboptimal solutions. On the other hand, a large population requires more evaluations per generation, possibly resulting in an unacceptably slow rate of convergence. In our experiments, the population size ranged from 300 to 500.

(3) Maximum generation (M_G):

As previously mentioned, one generation comprises the following steps: reproduction, crossover, mutation, permutation, evaluation, and some data collection procedures. This parameter is used to control an overall maximum number of generations to be run. For all of our experiments, the maximum generation is 50. Note that the maximum number of generations actually is 51 if the initial generation is considered.

(4) Crossover rate (R_c):

The crossover rate controls the frequency with which the crossover operator is applied. In each new generation, $R_c * P_s$ function models undergo crossover. The role of crossover is to introduce new function models into the population. If the crossover rate is too low, then the search may stagnate due to the lower exploration rate; the population may tend toward a stable selection of the best initial guesses. On the other hand, if the crossover rate is too high, the many superior individuals would quickly be crossed out of existence. The crossover rate in our experiments is in the range [0.8, 0.95]. Our experiments showed crossing 90 percent of the population works reasonably well, because good function models often have a high-enough fitness value to make their way into the noncrossed 10 percent.

(5) Mutation rate (R_m):

Mutation is a secondary search operator which increases the variability of the population. The fitness function drives the system toward better solutions, and sometimes it takes a wrong turn. The system will often converge after running and rerunning the fitness several times. A mutation operator can help move the process out of a niche. If the mutation rate is too high, the system starts declining in performance. For different implementations of mutation operation, the range of mutation rates is quite different. In the experiments we did with the second implementation method, the mutation rate was in the range [0.0, 0.2]. In most situations, we used the mutation rate as 0.01 or 0.02. For the first implementation of mutation operation, the range of the mutation rate could be larger than that of the second one. Note that in the experiments described in the next chapter, the mutation operation was performed using the second implementation method.

(6) Permutation rate (R_p):

The permutation operator is an extension of the inversion operation for string-based genetic algorithms to the domain of hierarchical genetic algorithm. The permutation operation can potentially bring closer together nodes of a relatively high fitness individual so that they are less subject to later disruption due to crossover (Koza 1990). However, our experience is that the benefits of this operation are purely potential and have yet to be observed for function identification problems. The permutation rate in our experiments ranged from 0 to 0.3.

(7) Convergent value (C_v):

This parameter serves as the convergence condition of the system. Usually we chose the convergent value C_v to be 0.99. This means that the system is successful in solving the given problem if at least one individual function model has the average of absolute differences between the predicted values of the model $f(x, B)$ and the given values of observations within 1% of the average of absolute values of observations (see equation (3.3)). For some problems, we chose the convergent value C_v as 0.999.

(9) Maximum depth of initial function models (D_i):

This parameter defines the maximum depth of initial function models. In our experiments, this parameter was set to be 4 or 5.

(10) Maximum depth of function models (D_m):

This parameter defines the maximum depth of function models generated by the genetic system. This limit prevents large amounts of computer time being expended on the extremely complicated function models. It was always set as 15 in our experiments.

(11) Options ($\mathbf{O} = \{e, o, s\}$):

(a) The option of the "elitist" selection strategy (e):

The elitist strategy stipulates that the individual with the best fitness in the current generation always survives intact to next generation. In the absence of this strategy, it is possible that the best individual disappears, due to crossover, mutation, or permutation.

(b) The option of LM optimization (o):

The HGSFI allows users to choose whether the system includes the optimizer (the Levenberg-Marquardt nonlinear regression algorithm) for a particular running in order to compare the two approaches.

(c) The option of simplifying function model (s):

If this option is chosen, the system variable SimplifyFlag is set as TRUE. This means that during each iteration after processing genetic operations, the models of the modified individuals will be simplified using the set of simplification rules. If the option is absent, the function models are simplified only in the final stage -- outputing function models to a file or on the screen.

Let HGSFI = ($\mathbf{S}$, P_s, M_G, R_c, R_m, R_p, C_v, D_i, D_m, $\mathbf{O}$) represent a particular running of the HGSFI with the settings of control parameters for the given problem. For example,

HGSFI = ({ c, x, +, -, *, /, exp, log, sin, cos},
500, 50, 0.9, 0.02, 0.1, 0.99, 4, 15, {e, o})

indicates one running of the HGSFI with

(a) The function table

S = { c, x, +, -, *, /, exp, log, sin, cos }

where c is a random constant and x is the independent variable;

(b) Population size is 500;

(c) Maximum number of generations is 50;

(d) Crossover rate is 0.9;

(e) Mutation rate is 0.02;

(f) Permutation rate is 0.1;

(g) Convergent value is 0.99;

(h) Maximum depth of initial function models is 4;

(i) Maximum depth of function models is 15;

(j) The options are { e, o }, which means this particular run of the HGSFI includes the elitist selection strategy and the Levenberg-Marquardt optimization.

4. TEST AND EVALUATION

We now describe the experiments of the HGSFI machine function identification system. These experiments were run under the UNIX operating system on DEC-5500 system. The HGSFI was compiled using gcc (GNU project C Compiler) version 1.36. These experiments were designed to test and evaluate the performance of the HGSFI. From the outset, generality has been a motivating force in the design of the HGSFI. Consequently, it seems appropriate that the system be tested and evaluated in a wide range of task domains. First we did some experiments to compare the system performance with and without the Levenberg-Marquardt optimization. Section 4.1 presents the detail of this comparison. Then we chose two sets of task domains to test the HGSFI. Section 4.2 discusses the results of the experiments of linear regression model problems. The experimental results of nonlinear regression model problems will be presented in section 4.3.

For all of the test problems, we randomly generated initial function models. This learning system does not need any prior knowledge about function models of the given problems. The only prior knowledge needed is the primitive functions, such as addition, multiplication, exponentiation, etc., that are to be used in representing the population of function models.

4.1. COMPARISON OF THE SYSTEM PERFORMANCE WITH AND WITHOUT LEVENBERG-MARQUARDT OPTIMIZATION

This set of experiments was conducted to compare the system performance with and without the Levenberg-Marquardt optimization. The test function equations are listed in Table 3. It includes the "symbolic regression" task domains faced by Koza's "genetic programming paradigm" (Koza 1990), which are problems (4.1), (4.2), (4.3), and (4.5).

Table 3: The first set of test problems

Prob -lem	Function Equation	X range	Pts
4.1	$Y = X + X^2 + X^3 + X^4$	[-2.0,2.0]	100
4.2	$Y = \sin(X) + \cos(X) + X + X^2$	[-5.0,5.0]	150
4.3	$Y = \cos(X + X)$	[0.0, 6.0]	20
4.4	$Y = X^2 - X^7$	[-5.0,5.0]	150
4.5	$Y = 3.1416*X+2.718*X^2$	[-5.0,5.0]	100
4.6	$Y = 0.808162*\exp(-1.21*X)$	[-2.0,8.0]	100
4.7	$Y = -2.3+3.0*X+0.45*X^2-1.23*X^3$	[-5.0,5.0]	100
4.8	$Y = 21.10-19.81*\exp(-0.00177*X^{3.180})$	[0.0,14.0]	140

The first four function equations have no numerical coefficients, the next four function equations have numerical coefficients. The sample data (no noise) of the test problems are uniformly generated using the test function equations in the given X ranges with the given number of points. For above test problems, except for problem (4.8), we chose the following control parameters

HGSFI = ({c, x, +, -, *, /, exp, log, sin, cos},
500, 50, 0.9, 0.02, 0.0, 0.99, 4, 15, {e})

to run the HGSFI without the option of LM optimization. The control parameters for problem (4.8) without the Levenberg-Marquardt optimization were

HGSFI = ({c, x, +, -, *, /, **, sq, exp, log, sin, cos},
500, 50, 0.9, 0.02, 0.0, 0.99, 4, 15, {e}).

where sq(X) means X^2 and X**C means X^C. For the runs with the Levenberg-Marquardt optimization, the option of LM optimization should be added to the control parameters. The experimental results of these test problems with and without LM optimization are shown in Table 4 and 5, respectively.

We define that a run is successful for the given sample data set if at least one function model in the run satisfies the given system convergence condition within the maximum number of generations.

In the tables, the **total runs** column contains the total number of runs for the given problem. The **success rate** means the ratio of the number of successful runs to the total number of runs for a given problem. The converged generation for a successful run is the generation at which at least one function model satisfies the system convergence condition within the maximum number of generations. The **avg. gen. for suc. runs** means the average number of the converged

Table 4: Experimental results of the first set of test problems without LM optimization

Prob -lem	Tot -al run s	Suc- cess Rate (%)	Avg. gen. for suc. runs	CPU time per run (min)	Average test value for success runs	Average test value for all runs	Average best R squares for success runs
4.1	30	43.3	32.85	5.411	.998040	.965313	.999980
4.2	30	73.3	23.73	6.190	.996840	.987427	.999962
4.3	25	100	3.36	0.048	.999633	.999633	.999997
4.4	30	0.0	-	4.274	-	.449476	-
4.5	30	56.7	32.17	6.702	.992964	.983683	.999894
4.6	30	50.0	25.47	8.549	.994589	.976314	.999965
4.7	30	0.0	-	10.82	-	.919131	
4.8	30	0.0	-	2.825	-	.890589	-

Table 5: Experimental results of the first set of test problems with LM optimization

Prob -lem	Tot -al run s	Suc- cess rate (%)	Avg. gen. for suc. runs	CPU time per run (min)	Average test value for success runs	Average test value for all runs	Average best R squares for success runs
4.1	26	100	10.5	37.58	.998411	.998411	.999994
4.2	20	100	10.65	79.33	.998450	.998450	.999983
4.3	30	100	1.066	0.425	.999952	.999952	1.00000
4.4	30	80.0	31.13	84.44	.997754	.923638	.999957
4.5	20	100	3.65	6.94	.999929	.999929	1.00000
4.6	20	100	2.05	3.92	.999880	.999880	.999994
4.7	50	100	10.44	30.10	.998385	.998385	.999992
4.8	49	100	14.31	107.4	.993469	.993469	.999791

generations for successful runs. Note that we define generation counter starts from 1 instead of 0 when we calculate the average number of the converged generations for successful runs. That is, initial generation is 1, the first generation is 2, and so on. The **CPU time per run** means the average of CPU time for all runs. The average test values for success runs and all runs are the averages of convergence test values (equation 3.3) of best function models for successful runs and all runs, respectively. The average best R squares for success runs is the average of R squares of best function models for successful runs.

For problems (4.1), (4.2), and (4.3) which are simple equations without numerical coefficients, the machine function identification system HGSFI without the option of LM optimization can find the highly fit function models, which are same as the function equations generated the sample data. For example, the highly fit function models that the HGSFI found for the test problems (4.1), (4.2), and (4.3) in some runs are as follows,

(1) y = (x+(((((x*x)*x)+(x*x))*x)+(x*x))) at generation 9,

(2) y = ((sin(x)+((x*x)+x))+cos(x)) at generation 26,

(3) y = cos((x+x)) at generation 0,

respectively. The success rate for problems (4.2) and (4.3) is high. Note that the SSResid (see equation 2.16) for these correct function models the HGSFI found is not equal to zero because of roundoff error.

For problems (4.5) and (4.6) which are simple equations with numerical coefficients, the system without the option of LM optimization can still find a reasonably good fit function model to the given sample data set, at relatively high success rate. For example, the success rate in 30 runs for problem (4.6) without the option of nonlinear regression is 50.0 percent. The summary statistics and a highly fit function model the HGSFI found in a run without the option of LM optimization is shown in Figure 19. The highly fit function model the HGSFI found in the run is equivalent to

$$y = 0.8073855e^{-1.211x}$$

which is very close to the equation (4.6). This result shows that the genetic algorithm can do pretty good job for some "constant discoveries".

Note that the meanings of the columns in Figure 19 are as follows:

(1) Gens stands for the generation number;

(2) Trials is the number of individuals evaluated;

(3) The **online** performance is defined as the average of the normalized performance of all evaluated individuals over the course of the genetic search.

(4) The **offline** performance is defined as the average of the normalized performance of best individuals over

```
##### HGSFI Summary Report #####
### Best Function Model ###
# Y=EXP((((C1-X)*C2)-X))

num_of_const = 2
 C1          C2
-1.014e+00  2.110e-01

### Summary Statistics for the Best Function Model ###
   Source              DF   Sum of Squares   Mean Squares
   Residual            98     2.42873e-04    2.47830e-06
   Uncorrected Total  100     3.84288e+02
   Corrected Total     99     3.20693e+02
   R_squared = 1 - Residual SS / Corrected SS = 0.9999992

Sum of absolute (Yi - Yi'): 5.54913e-02
        Sum of absolute Yi: 7.97468e+01
ConvergentTestValue = 1 - SumOfAbsDy / SumOfAbsY = 0.9993042
( Success condition: ConvergentTestValue > 0.9900 )

Gens Trials Online Offline Best   Average    Current Best
                           Performance   SSResid    Rsquared
  0    500   .00465 .02412 .024118 .004649 3.657e+01  .885958
  1    956   .00576 .02412 .024118 .007172 3.657e+01  .885958
  2   1407   .00643 .04207 .077975 .008023 3.631e+00  .988677
  3   1861   .00691 .05211 .082230 .008574 3.465e+00  .989196
  4   2314   .00782 .05864 .084778 .011758 4.027e+00  .987444
  5   2766   .00898 .08193 .198345 .015764 1.178e+00  .996328
  6   3220   .01111 .10029 .210478 .025507 5.747e-01  .998208
  7   3677   .01395 .15413 .530978 .036361 4.953e-02  .999846
  8   4135   .01657 .19600 .530978 .042642 4.953e-02  .999846
  9   4589   .01914 .22950 .530978 .049673 4.953e-02  .999846
 10   5045   .02132 .25690 .530978 .053639 4.953e-02  .999846
 11   5502   .02313 .27974 .530978 .050640 4.953e-02  .999846
 12   5961   .02447 .29907 .530978 .050649 4.953e-02  .999846
 13   6424   .02584 .31563 .530978 .053312 4.953e-02  .999846
 14   6880   .02740 .32999 .530978 .059563 4.953e-02  .999846
 15   7339   .02861 .34255 .530978 .056663 4.953e-02  .999846
 16   7801   .02976 .35364 .530978 .057022 4.953e-02  .999846
 17   8259   .03069 .36349 .530978 .058276 4.953e-02  .999846
 18   8723   .03195 .37230 .530978 .059776 4.953e-02  .999846
 19   9182   .03330 .38024 .530978 .065697 4.953e-02  .999846
 20   9650   .03531 .40726 .947426 .079663 2.429e-04  .999999
```

Figure 19: The summary report of problem (4.6) in a run without LM optimization

the course of the genetic search.

(5) Best performance and average performance are the best normalized performance and the average normalized performance in the current generation, respectively.

(6) Current best SSResid and R squared are the sum of squared residuals and R squared of the best individual in the current generation, respectively.

For the complicated problems such as problems (4.4), (4.7), and (4.8), however, the system without the option of LM optimization cannot find a function model which satisfies the given system convergence condition within the given maximum number of generations. For example, the success rates in 30 runs for problems (4.4), (4.7), and (4.8) are zero. With the option of LM optimization, the system can find the function models, which satisfy the system convergence conditions, for the given sample data points of these problems. Figure 20 is the summary report of one run of the HGSFI with LM optimization for problem (4.7). After simplification, the most fit function model the system found in the run is equivalent to

$$y = -2.3+3.0*X+0.45*X^2-1.23*X^3 .$$

```
##### HGSFI Summary Report #####
### Best Function Model ###
# Y=((((X*X)*(C1*X))+C2)+C3-(((X*X)C4*)+((X*C5)-C6))))

num_of_const = 6
 C1           C2           C3           C4           C5
-1.230e+00   4.596e+00   -5.138e+00   -4.500e-01   -3.000e+00
 C6
-1.758e+00
Uncertainties:
 5.295e-03   4.472e+02   3.162e+02   1.344e-02   8.661e-02
 3.162e+02

### Summary Statistics for the Best Function Model ###
   Source               DF   Sum of Squares   Mean Squares
   Residual             94     6.40283e-27    6.81152e-29
   Uncorrected Total   100     2.57184e+05
   Corrected Total      99     2.56378e+05
   R_squared = 1 - Residual SS / Corrected SS = 1.0000000

Sum of absolute (Yi - Yi'): 4.09665e-13
        Sum of absolute Yi: 3.20289e+03
ConvergentTestValue = 1 - SumOfAbsDy / SumOfAbsY = 1.0000000
( Success condition: ConvergentTestValue > 0.9900 )

Gens Trials Online Offline Best   Average    Current Best
                           Performance    SSResid   Rsquared
  0   500    .00019 .00055 .000546 .000194 4.962e+04  .806465
  1   952    .00023 .00056 .000580 .000269 1.039e+05  .594881
  2  1404    .00025 .00057 .000584 .000291 1.035e+05  .596337
  3  1857    .00025 .00057 .000584 .000275 1.035e+05  .596337
  4  2313    .00026 .00058 .000584 .000290 1.035e+05  .596337
  5  2766    .00027 .00060 .000710 .000301 3.177e+04  .876084
  6  3222    .00027 .00083 .002242 .000305 2.640e+03  .989703
  7  3674    .00028 .00102 .002309 .000314 2.414e+03  .990586
  8  4126    .00028 .00116 .002309 .000334 2.414e+03  .990586
  9  4579    .00029 .00135 .003023 .000372 1.454e+03  .994330
 10  5033    .00030 .00162 .004365 .000475 7.845e+02  .996940
 11  5486    .00033 .00185 .004365 .000711 7.845e+02  .996940
 12  5944    .00055 .07863 1.00000 .003088 6.403e-27 1.000000
```

Figure 20: The summary report of problem (4.7) in a run with LM optimization

If we define "the correct function model" the HGSFI found for the given problem as

(1) after simplification, the form of "the correct function model" is same as the form of the function

equation generated the sample data, except for the difference of some constant term, and

(2) its R squares > 0.999999,

then the success rates for finding the correct function models of problems (4.1) to (4.8) by HGSFI with and without LM optimization are listed in Table 6.

Table 6: The success rate for finding the correct function models of the first set of test problems

Problem	Total runs without LM	Success rate without LM option	Total runs with LM	Success rate with LM option
4.1	30	30.00%	26	53.85%
4.2	30	30.00%	20	60.00%
4.3	25	96.00%	30	100%
4.4	30	0.00%	30	23.33%
4.5	30	3.33%	20	95.00%
4.6	30	3.33%	20	85.00%
4.7	30	0.00%	50	56.00%
4.8	30	0.00%	49	0.00%

Comparing the system performance with and without the option of LM optimization, the average convergence test value of all runs and the average R squares of success runs with nonlinear regression are better than those without nonlinear regression. The success rate with LM optimization is much higher than that without LM optimization except in some extremely simple problems such as problem (4.3). In addition, the average number of the converged generations for successful runs with LM optimization is less than that without LM optimization. In other words, the system performance with the Levenberg-Marquardt optimization is better than that without the Levenberg-Marquardt optimization. However, the average CPU time to solve the given problems with LM optimization is longer than that without LM optimization, because the optimizer spends a lot of time to optimize the coefficients of function models.

4.2. LINEAR REGRESSION MODEL PROBLEMS

Linear regression model is the model in which all the parameters (i.e. estimated coefficients) appear linearly. This does not mean the response variable Y has the linear relationship with independent variable X. For example, the model

$$Y_t = C_0 + C_1 X_t + C_2 X_t^2 + \varepsilon_t$$

is a linear regression model instead of a nonlinear regression model (because Y_t is nonlinear in X_t). For this set of experimental problems, we use the function equations in Table 7 to generate the corresponding experimental data (no noise).

For this set of test problems, we chose the following control parameters:

HGSFI = ({c, x, +, -, *, /, exp, log, sin, cos},
500, 50, 0.9, 0.02, 0.0, 0.99, 4, 15, {e, o}).

Note that LM optimization was included on all runs for this set of test problems. The experimental results of the second set of test problems are listed in Table 8.

Table 7: The second set of test problems

Prob -lem	Function Equation	X range	Pts
4.7	$Y = -2.3+3.0*X+0.45*X^2-1.23*X^3$	[-5.0,5.0]	100
4.9	Y = 3.2*sin(X)+2.5*cos(X)	[-2.0, 18]	200
4.10	Y = 0.45+6.032*exp(-X)	[0.0,20.0]	100
4.11	Y = 0.1-2*X+0.5*log(X)	(0.0,20.0]	100

Table 8: Experimental results of the second set of test problems with LM optimization

Prob -lem	Tot -al run s	Suc-cess Rate (%)	Avg. Gen. for suc. runs	CPU time per run (min)	Average test value for success runs	Average test value for all runs	Average best R squares for success runs
4.7	50	100	10.44	30.10	.998385	.998385	.999992
4.9	20	100	4.8	16.62	.999776	.999776	1.00000
4.10	23	95.7	10.36	95.75	.999231	.991547	.999993
4.11	25	100	1.0	2.04	.996607	.996607	.999830

Problem (4.7) is a third-degree polynomial model in one independent variable. The degree (or order) of an individual term in a polynomial is defined as the sum of the powers of the independent variable in the term. The degree of the entire polynomial is defined as the degree of the highest-degree term. All polynomial models, regardless of their degree, are linear in the parameters. Higher-degree polynomial models are regarded in most situations as approximations to the true models. Figure 20 shows the statistical report of one run for problem (4.7).

```
##### HGSFI Summary Report #####
### Best Function Model ###
# Y=(C1*COS((X+C2)))

num_of_const = 2
 C1          C2
 4.061e+00  -9.076e-01

Uncertainties:
 9.998e-02   2.468e-02

### Summary Statistics for the Best Function Model ###
   Source             DF   Sum of Squares   Mean Squares
   Residual          198     1.66727e-13    8.42055e-16
   Uncorrected Total 200     1.65295e+03
   Corrected Total   199     1.64781e+03
   R_squared = 1 - Residual SS / Corrected SS = 1.0000000

Sum of absolute (Yi - Yi'): 5.01913e-06
        Sum of absolute Yi: 5.19388e+02
ConvergentTestValue = 1 - SumOfAbsDy / SumOfAbsY = 1.0000000
( Success condition: ConvergentTestValue > 0.9900 )

Gens Trials Online Offline Best   Average    Current Best
                           Performance   SSResid   Rsquared
  0   500   .00109 .00238 .002381 .001093 1.072e+03  .349228
  1   953   .00133 .00281 .003234 .001607 5.969e+02  .637759
  2  1405   .00215 .33520 .999995 .003694 1.667e-13 1.000000
```

Figure 21: The summary report of problem (4.9) in a run with LM optimization

Problem (4.9) is a periodic model. The summary report and the highly fit function model the system found in a run is shown in Figure 21. It is equivalent to

y = 3.20*sin(X) + 2.50*cos(X).

```
##### HGSFI Summary Report #####
### Best Function Model ###
# Y=((C1/EXP(X))+C2)

num_of_const = 2
 C1          C2
 6.032e+00   4.500e-01
Uncertainties:
 6.053e-01   1.054e-01

### Summary Statistics for the Best Function Model ###
   Source              DF   Sum of Squares   Mean Squares
   Residual            98     7.47844e-14    7.63106e-16
   Uncorrected Total  100     1.60564e+02
   Corrected Total     99     9.92914e+01
   R_squared = 1 - Residual SS / Corrected SS = 1.0000000

Sum of absolute (Yi - Yi'): 2.30815e-06
        Sum of absolute Yi: 7.82765e+01
ConvergentTestValue = 1 - SumOfAbsDy / SumOfAbsY = 1.0000000
( Success condition: ConvergentTestValue > 0.9900 )

Gens Trials Online Offline Best   Average    Current Best
                          Performance   SSResid   Rsquared
  0   500   .00731 .02746 .027458 .007314 9.944e+01 -.001525
  1   957   .00941 .03955 .051652 .012038 4.333e+00  .956366
  2  1411   .01124 .35970 .999998 .015128 7.478e-14 1.000000
```

Figure 22: The summary report of problem (4.10) in a run with LM optimization

Problem (4.10) is an exponential (decay) model. The model (4.11) is the linear form of Hoerl's special function $Y = aX^b e^{cX}$ ($\ln Y = \ln a + b (\ln X) + cX$). The summary reports of problems (4.10) and (4.11) in some run are given in Figure 22 and Figure 23, respectively. The equivalent equations of the highly fit function models the system found are

y = 0.45 + 6.032*exp(-X)

and

y = 0.1 - 2.0*X + 0.5*log(X),

respectively.

```
##### HGSFI Summary Report #####
### Best Function Model ###
# Y=(((C1+(X-C2))/C3)-(C4*LOG(X)))

num_of_const = 4
 C1           C2           C3           C4
 9.056e-01   9.556e-01   -5.000e-01   -5.000e-01
Uncertainties:
 3.162e+02   3.162e+02   2.532e-03    1.089e-01

### Summary Statistics for the Best Function Model ###
   Source               DF    Sum of Squares   Mean Squares
   Residual             96      8.84624e-14    9.21483e-16
   Uncorrected Total   100      3.10789e+05
   Corrected Total      99      8.05437e+04
   R_squared = 1 - Residual SS / Corrected SS = 1.0000000

Sum of absolute (Yi - Yi'): 2.62445e-06
        Sum of absolute Yi: 4.79839e+03
ConvergentTestValue = 1 - SumOfAbsDy / SumOfAbsY = 1.0000000
( Success condition: ConvergentTestValue > 0.9900 )

Gens Trials Online Offline Best   Average    Current Best
                          Performance   SSResid   Rsquared
  0   500   .00517 1.0000 .999997 .005166 8.846e-14 1.000000
```

Figure 23: The summary report of problem (4.11) in a run with LM optimization

Table 9: The success rate for finding the correct function models of the second set of test problems

Problem	4.7	4.9	4.10	4.11
Total runs	50	20	23	25
Success rate with LM option	56.00%	85.00%	82.61%	8.0%

The success rates for finding the correct function models of the second set of problems by HGSFI with LM optimization are listed in Table 9. Note that the success rate for finding the correct function models of problem 4.11 is quite low. But if we increase the convergent value to 0.999 to run the HGSFI for problem 4.11, the success rate will be higher than 8.0%.

For almost all of the linear regression model problems we tested, the HGSFI can rapidly converge on a highly fit function model which satisfies the given system convergence condition within the maximum number of generations. Note that due to roundoff error the **SSResid**s of the highly fit function models that the HGSFI found for this set of test problems are not exactly equal to zero.

4.3. NONLINEAR REGRESSION MODEL PROBLEMS

A nonlinear regression model is one in which at least one of its parameters appears nonlinearly, for example

$$Y_k = \alpha X_k^{\beta} + \varepsilon_k \ .$$

In the formal sense, nonlinear means that at least one of the derivatives of Y_k with respect to α and β is a function of at least one of those parameters. In above equation, the derivative of Y_k with respect to α and the derivative of Y_k with respect to β are both functions of α and/or β, so that this model is a nonlinear regression model. For the third set of test problems, we use the following nonlinear regression equations to generate the experimental data (no noise):

Problem (4.8): Weibull-type model

$$y = \alpha - \beta e^{-\gamma x^{\delta}}$$

where α = 21.10, β = 19.81, γ = 0.00177, and δ = 3.180. The range of x is [0.0, 14.0]. The number of sample data points is 140. This nonlinear statistical model is widely used in biology and economics, which produces sigmoidal or "S-shaped" growth curve. The parameter values of this model are abstracted from Ratkowsky's book "Nonlinear Regression Modeling" pp 65 (Ratkowsky 1983).

Problem (4.12): Inverse polynomial model

$$y = \frac{x}{\alpha+\beta x}$$

where α = 0.3 and β = 0.06. The range of X is [0.0, 10.0]. The number of sample data points is 100. This function is a monotonically increasing function of X that very slowly approaches the asymptote Y = 1/β.

Problem (4.13): Logistic growth model

$$y = \frac{\alpha}{1.0 + \gamma e^{\beta x}}$$

where α = 10.0, β = -0.8, and γ = 25.0. The range of X is [0.0, 10.0]. The number of sample data points is 100. The logistic growth model is frequently used in biology, agriculture, and engineering. This function gives the "S-shaped" growth curve starting at Y = $\alpha/(1+\gamma)$ at X = 0 and approaching the asymptote Y = α as X gets large. Note that problems (4.12) and (4.13) are abstracted from Rawlings' book "Applied Regression Analysis" pp 302 (Rawlings 1988).

Problem (4.14): Two term exponential model

$$y = \frac{C_1}{C_1 - C_2}(e^{-C_2 x} - e^{-C_1 x})$$

where $C_1 = 0.6$ and $C_2 = 0.3$. The range of X is [0.0, 20.0]. The number of sample data points is 100. This function is abstracted from Rawlings' book "Applied Regression Analysis" pp 391 (Rawlings 1988).

Table 10: Experimental results of the third set of test problems with LM optimization

Prob -lem	Tot -al run s	Suc-cess Rate (%)	Avg. Gen. for suc. runs	CPU time per run (min)	Average test value for success runs	Average test value for all runs	Average best R squares for success runs
4.8	49	100	14.31	107.4	.993469	.993469	.999791
4.12	20	100	8.4	99.84	.999674	.999674	.999996
4.13	30	96.7	18.55	350.3	.999344	.998610	.999998
4.14	20	95.0	11.42	152.5	.995469	.994697	.999951

The settings of control parameters for problems (4.12) and (4.13) are

HGSFI = ({c, x, +, -, *, /, **, sq, exp, log, sin, cos}, 500, 50, 0.9, 0.02, 0.0, 0.999, 4, 15, {e, o}). The only difference of the settings of control parameters to run the HGSFI for problems (4.8) and (4.14) is the convergent value. The latter uses 0.99 instead of 0.999. The summary of experimental results of these test problems is shown in Table 10.

This set of experimental problems is very difficult and challenge for the current implementation of the HGSFI. Because function discovery and nonlinear regression are both global optimization problems. To discover the "correct" function model and the "correct" values of the coefficients of the function model, the system needs to search large function space and large constant space. The two together make the problem very hard. In addition, so far there are not any efficient global optimization algorithms for nonlinear regression. The HGSFI implementation uses the Levenberg-Marquardt optimization algorithm to find the "best" fit coefficients of function models for the given sample data set, which is a local optimization algorithm. If a function model is "correct" for the given problem but the starting values of the coefficients of the function model are not around the global optimizing point, then the poor starting values may result in convergence to an unwanted stationary point of the sum of squares surface (Draper and Smith 1981). This unwanted point may have coefficient values which do not provide the true minimum value of S(B) (equation 2.3). Then the "correct" function model may be thrown away due to the poor fitness.

```
##### HGSFI Summary Report #####
### Best Function Model ###
# Y=(((C1+X)/(C2+X))+C3)

num_of_const = 3
 C1            C2            C3
-7.8333e+01   5.0000e+00    1.5667e+01
Uncertainties:
 1.0730e+01   6.6713e-01    4.7124e-01

### Summary Statistics for the Best Function Model ###
   Source              DF    Sum of Squares   Mean Squares
   Residual            97      8.10634e-14    8.35705e-16
   Uncorrected Total  100      1.24026e+04
   Corrected Total     99      1.11560e+03
   R_squared = 1 - Residual SS / Corrected SS = 1.0000000

Sum of absolute (Yi - Yi'): 2.44214e-06
        Sum of absolute Yi: 1.06240e+03
ConvergentTestValue = 1 - SumOfAbsDy / SumOfAbsY = 1.0000000
( Success condition: ConvergentTestValue > 0.9990 )

Gens Trials Online Offline Best   Average    Current Best
                           Performance    SSResid   Rsquared
  0   500   .00076 .01197 .011970 .000764 9.641e+01  .913581
  1   956   .00123 .01683 .021690 .001816 3.247e+01  .970898
  2  1414   .00169 .01914 .023750 .002947 2.415e+01  .978349
  3  1869   .00206 .02307 .034877 .003640 1.047e+01  .990617
  4  2327   .00258 .03146 .065009 .005181 3.873e+00  .996528
  5  2783   .00308 .04403 .106887 .006788 1.004e+00  .999100
  6  3239   .00379 .05419 .115118 .010114 9.501e-01  .999148
  7  3706   .00523 .17241 .999998 .017161 8.106e-14 1.000000
```

Figure 24: The summary report of problem (4.12) in a run with LM optimization

For these nonlinear regression model problems, the HGSFI can still quickly find high performance function models to fit the given sample data points with very high success rate. Sometimes it can find the highly fit function model which is same as the function equation generated the sample data set. For example, the "best" function model the HGSFI found for problem (4.12) in a run is

$$y = \frac{x}{0.30000+0.06000x}$$

which is equivalent to the function equation (4.12). The

```
##### HGSFI Summary Report #####
### Best Function Model ###
# Y=((C1-(C2*X))-(((C3*(C4-X))-(((C5-X)*(C6*X))*
     (X+C7)))*(X*X)))

num_of_const = 7
 C1          C2          C3          C4          C5
 1.257e+00  -3.169e-01   5.379e-02   7.351e+00   2.259e+01
 C6          C7
-5.980e-04  -1.016e+01
Uncertainties:
 4.770e-01   7.021e-01  2.314e+00   3.162e+02   3.111e+02
 1.315e-04   3.111e+02

### Summary Statistics for the Best Function Model ###
   Source               DF    Sum of Squares   Mean Squares
   Residual             133     1.27622e+00    9.59562e-03
   Uncorrected Total    140     2.79300e+04
   Corrected Total      139     8.64718e+03
   R_squared = 1 - Residual SS / Corrected SS = 0.9998524

Sum of absolute (Yi - Yi'): 1.10132e+01
        Sum of absolute Yi: 1.64304e+03
ConvergentTestValue = 1 - SumOfAbsDy / SumOfAbsY = 0.9932970
( Success condition: ConvergentTestValue > 0.9900 )

Gens Trials Online Offline Best   Average    Current Best
                            Performance   SSResid   Rsquared
  0   500   .00061 .00723 .007230 .000611 1.764e+02  .979602
  1   955   .00076 .00839 .009540 .000961 1.161e+02  .986569
  2  1406   .00095 .01019 .013802 .001443 6.239e+01  .992785
  3  1858   .00126 .01286 .020871 .002394 2.412e+01  .997211
  4  2315   .00155 .01446 .020871 .003029 2.412e+01  .997211
  5  2767   .00181 .01553 .020871 .003677 2.412e+01  .997211
  6  3228   .00218 .01766 .030419 .004940 1.374e+01  .998410
  7  3695   .00243 .02586 .083241 .005016 1.276e+00  .999852
```

Figure 25: The summary report of problem (4.8) in a run with LM optimization

summary report of that run is shown in Figure 24. In most situations, however, the HGSFI can find a higher-degree polynomial function model to fit the given sample data points. Figure 25 is an example for problem (4.8). The reasonably good fit function model the HGSFI found in that run is equivalent to

$$y = 1.257+0.3169X-0.39541X^2+0.19104X^3-0.01958X^4+0.000598X^5 \text{ .}$$

The success rates for finding the correct function models of the third set of problems by HGSFI with LM optimization are listed in Table 11.

Table 11: The success rate for finding the correct function models of the third set of test problems

Problem	4.8	4.12	4.13	4.14
Total runs	49	20	30	20
Success rate with LM option	0.00%	20.00%	0.00%	0.00%

5. FUTURE RESEARCH

The experimental results of this research has clearly established the feasibility of the HGSFI design. However, it is worthwhile to do further research for the following problems. Firstly, the further work might include studying other selection strategies for reproduction in the genetic algorithm. For instance, the selection for mating can be distributed such as in the parallel genetic algorithm (Mühlenbein 1991). In Mühlenbein's parallel genetic algorithm, the selection is divided into a mate selection step and an acceptance step. A population is divided into several groups. (For example, individuals 1 to 10 are in group 1, individuals 11 to 20 are in group 2, and so on. Note that Mühlenbein's implementation is more complicated than this.) In the mate selection step, each individual selects a partner in the neighborhood (group) for mating. In the acceptance step, the offspring will replace its parent, if it is better than the parent. Each individual may improve its fitness during its lifetime by local hill climbing. Mühlenbein applied the parallel genetic algorithm (PGA) to the traveling salesman problem, the autocorrelation problem and the graph partitioning problem. "In all these problems, the PGA has found solutions of very large problems, which are comparable or even better than any other solution found by other heuristics." (Mühlenbein 1991).

It would be useful to study the effects of varying the rates at which the genetic operators are applied. The system should learn better values for control parameters. Hopefully, this would yield results concerning the adaptive and optimal settings of these parameters for the given problem during the learning process.

Further work might add some general heuristic rules into the system to guide the generation of initial function models and the breeding of the new function models. For instance, we may consider the physical unit consistency of the given data set and the function model to guide function model generation such as in ABACUS (Falkenhainer and Michalski 1986). Units analysis enables one to greatly reduce the size of the search space by examining the compatibility of variables' units. And we might include other statistic analysis techniques to get more information from the given sample data set to guide function model generation.

Nonlinear regression is a global optimization problem. Global optimization, however, is a difficult area in its own right. So far there are not any practical and efficient global optimization techniques for nonlinear regression. The HGSFI uses the Levenberg-Marquardt nonlinear regression algorithm to find the "best" fit coefficients of function models to the given problem. The Levenberg-Marquardt algorithm is a local optimization algorithm. Unfortunately, the results of nonlinear regression often depend on having good starting values for the coefficients to be estimated. The poor starting values may result in convergence to an unwanted stationary point of the sum of squares surface, which is not the true minimum value of S(B). Then the "correct" function model may be thrown away thanks to the poor fitness. In order to overcome the above weakness, the further research might add a general method to set the starting values of the coefficients of a function model according to the given problem.

The new system might use multiple optimization techniques to optimize the values of the coefficients of a function model because one algorithm may perform better than the other for a particular problem.

6. SUMMARY AND CONCLUSIONS

This thesis has focused on the problem of constructing a machine function identification system to find a highly fit function model for the given sample data points. We began by developing the general hierarchical representation scheme of the system's knowledge (function models). The hierarchical representation can represent arbitrary function models whose size, shape, and complexity can dynamically change during the learning process. Then we discussed the selection of domain independent learning methods for manipulating the system knowledge (function models). The learning component of the machine function identification system takes advantage of two kinds of domain independent optimization procedures: the hierarchical genetic algorithm and the Levenberg-Marquardt nonlinear regression algorithm. A system called HGSFI was implemented using this design method. Utilizing the initial function models generated at random, HGSFI searches the space of function models, dynamically creates new generation of function models using Darwinian principles of reproduction and survival of the fittest. It optimizes the coefficients of the function models using the Levenberg-Marquardt nonlinear regression algorithm, and tries to make the function models "best" fit the given sample data set. Through the iterative learning process, the individual function models in the new generation inherit the parents' merits, adjust themselves to environmental fluctuation, and enable themselves to cope with the environment and to continue their existence. They are getting better and better to fit the given sample data points. When some function model satisfies the criterion of the system performance, the system returns the most fit function model the system has found as the solution of the given problem and halts.

This machine function identification system, HGSFI, does not need any prior knowledge about the function model of the given problem. The only prior knowledge it needs is the set of function operations and terminals which should be capable of solving the problem (i.e. some composition of the available function operations and terminals should yield a solution).

Having finalized the design, a series of experiments were conducted to investigate the operative characteristics of the HGSFI implementation. The experimental results of the first set of test problems show that the system performance with the nonlinear regression optimizer is much better than that without the nonlinear regression optimizer (as measured by success rate). As a demonstration of the feasibility of the HGSFI design as a machine function identification system, experiments were conducted in a wide range of task domains. The test problems are grouped in two categories of function identification problems: linear regression model problems and nonlinear regression model problems. Initialized in each test with randomly generated initial function models, the HGSFI implementation is shown to rapidly converge on highly fit function models in the both task domains.

BIBLIOGRAPHY

Ayala, Francisco J. and Valentine, James W. (1979), **Evolving: the Theory and Processes of Organic Evolution**, Menlo Park, CA: Benjamin/Cummings

Daniel, Cuthbert, Wood, Fred S. and Gorman, John W. (1980), **Fitting Equations To Data**, New York: John Wiley & Sons

Davis, Lawrence (1987), ed., **Genetic Algorithms and Simulated Annealing**, Los Altos, CA: Morgan Kaufmann

Draper, N.R. and Smith, H. (1981), **Applied Regression Analysis**, 2nd ed., New York: John Wiley & Sons

Falkenhainer, Brian C. and Michalski, Ryszard S. (1986), **Integrating Quantitative and Qualitative Discovery: the ABACUS System**, Machine Learning 1, 367-401

Gallant, A. Ronald (1987), **Nonlinear Statistical Models**, New York: John Wiley & Sons

Goldberg, David E. (1989), **Genetic Algorithms in Search, Optimization, and Machine Learning**, Reading, Massachusetts: Addison-Wesley

Grefenstette, John J. (1984), **GENESIS: A System for Using Genetic Search Procedures**, Proceedings of the 1984 Conference on Intelligent System and Machines

Grefenstette, John J. (1987), ed., **Proceedings of the Second International Conference on Genetic Algorithms**, Cambridge, Massachusetts: Lawrence Erlbaum

Holland, John H. (1975), **Adaptation in Natural and**

Artificial Systems, Ann Arbor, MI: University of Michigan Press

Kagiwada, Harriet H. (1974), **System Identification: Methods and Applications**, Reading, Massachusetts: Addison-Wesley

Kennedy, W. J. and Gentle, J.E. (1980), **Statistical Computing**, New York: Dekker

Koza, John R. (1989), **"Hierarchical Genetic Algorithms Operating on Populations of Computer Programs,"** Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI), San Mateo, CA: Morgan Kaufmann

Koza, John R. (1990), **Genetic Programming: a Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems**, Technical Report STAN-CS-90-1314, Computer Science Department, Stanford University

Langley, P., Bradshaw, G.L., and Simon, H.A. (1983), **Rediscovering chemistry with the BACON system**, in R.S. Michlski, J.G. Carbonell, & T.M. Mitchell (Eds.), Machine Learning: An artificial intelligence approach, San Mateo, CA: Morgan Kaufmann

Langley, P., Simon, H.A., Bradshaw, G.L., and Zytkow, J.M. (1987), **Scientific Discovery: Computational Explorations of the Creative Processes**, Cambridge, MA: MIT Press

Montgomery, Douglas C. and Peck, Elizabeth A. (1982),

Introduction to Linear Regression Analysis, New York: John Wiley & Sons

Mühlenbein, Heinz (1991), **Evolution in Time and Space - The Parallel Genetic Algorithm**, in Gregory J.E. Rawlins (Eds.), Foundations of Genetic Algorithms, San Mateo, CA: Morgan Kaufmann

Nordhausen, Bernd and Langley, P. (1990), **An Integrated Approach to Empirical Discovery**, in J. Shrager & P. Langley (Eds.), Computational Models of Scientific Discovery and Theory Formation, San Mateo, CA: Morgan Kaufmann

Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. (1988), **Numerical Recipes in C: The Art of Scientific Computing**, New York: Cambridge University Press

Ratkowsky, David A. (1983), **Nonlinear Regression Modeling: A Unified Practical Approach**, New York: Marcel Dekker

Rawlings, John O. (1988), **Applied Regression Analysis: A Research Tool**, Pacific Grove, CA: Wadsworth & Brooks

Schaffer, J. David (1989), ed., **Proceedings of the Third International Conference on Genetic Algorithms**, San Mateo, CA: Morgan Kaufmann

Smith, S.F. (1980), **A Learning System Based on Genetic Adaptive Algorithms**, Ph.D. dissertation, Dept. of Computer Science, University of Pittsburgh

Wallace, Bruce and Srb, Adrian M. (1961), **Adaptation,** Englewood Cliffs, New Jersey: Prentice-Hall

Wright, Alden H. (1991), **Genetic Algorithms for Real Parameter Optimization**, in Gregory J.E. Rawlins (Eds.), Foundations of Genetic Algorithms, San Mateo, CA: Morgan Kaufmann

Zytkow, J. (1987), **Combining many searches in the FAHRENHEIT discovery system**, Proceedings of the Fourth International Workshop on Machine Learning (pp. 281-287), Irvine, CA: Morgan Kaufmann