Genetic Programming with Incremental Data Inheritance

Byoung-Tak Zhang
Artificial Intelligence Lab (SCAI)
Dept. of Computer Engineering
Seoul National University
Seoul 151-742, Korea
http://scai.snu.ac.kr/˜ btzhang

Je-Gun Joung
Artificial Intelligence Lab (SCAI)
Dept. of Computer Engineering
Seoul National University
Seoul 151-742, Korea
http://scai.snu.ac.kr/˜ jgjoung

Abstract

A data-driven method for accelerating genetic programming is presented. This method, called incremental data inheritance or IDI for short, evolves programs using program-specific subsets of given data which also evolve incrementally as generation goes on. The concept of data evolution in IDI is contrasted to conventional genetic programming in which all the given training data are used repeatedly. IDI is also distinguished from the previous subset selection methods in that each program in IDI evolves its own data set of incremental size rather than a common data set of fixed or arbitrary size for the whole population. The method has been applied to time series prediction. Compared to the conventional methods, IDI significantly reduced the evolution speed of genetic programming without loss of the generalization accuracy of evolved programs. We also provide a theoretical foundation of the IDI method from the Bayesian inference point of view.

1 Introduction

The computational time for evolving programs in genetic programming (GP) is proportional to the product of population size, generation number, and the size of data for fitness evaluation. Typical population size for GP ranges from a few hundreds to several thousands [7, 2]. A run usually takes several dozens to hundreds of generations. The data size depends on applications, ranging from a few dozens to thousands of fitness cases. Fitness evaluation takes the most of computational effort in GP since it requires programs to be executed against fitness cases.

The basic idea behind the incremental data inheritance (IDI) approach described in this paper is that fitness evaluation time can be reduced by evolving the programs using a small but representative set of fitness data rather than all the fitness cases geiven. We present a method for actively selecting (or evolving) data sets while evolving programs. This idea of active data selection in supervised learning is not very new; It has been used for efficient learning and optimization of neural networks in GENIE [12]. The given data set B is initially divided into two disjoint subsets of a small training data D and the remaining candidate data C. Then, in its selective incremental learning or SEL mode [17, 14], the neural network is trained on D which evolves incrementally by choosing data from C that have large errors on the current neural network. When the candidate set C is automatically generated by a genetic algorithm, the method is called creative incremental learning or CSEL [18]. When the network architecture also grows while the training data set grows, the method is called self-development learning or SELF [13]. In GENIE, only one program and one data set are maintained.

Recently, Gathercole and Ross [4, 5] introduced a similar data selection method into genetic programming. Their DSS (dynamic subset selection) method involves picking a subset of cases from the full training set for each generation. Each generation of GP is evaluated using only this subset. In DSS, the subset size is fixed as an algorithm parameter. The cases are selected according to their difficulty and age. Siegel [9] describes a similar algorithm, but does not make use of the age of data. Daida et al. [3] studied a dynamic fitness evaluation method in genetic programming for image classification.

More recently, Teller and Andre [11] presented the RAT (rational allocation of trials) method that automatically chooses the number of fitness cases. RAT allocates individual programs to tournaments prior to their evaluation, and then allocates fitness cases only to those individuals for which the cost of evaluating another fitness case is outweighed by the expected utility that the new information will provide. Previously, Angeline and Pollack [1] reported that competitive environments evolve better solutions for complex tasks.

Incremental data inheritance is distinguished from the previous studies in several aspects. First, IDI maintains a separate data set associated with each program, rather than a common data set for all programs as in DSS. Second, the data size in IDI increases monotonically as generation goes on. This is important for convergence of performance and contrasted with RAT. Third, data sets evolve by inheriting the data of their parents (data inheritance) just as the programs inherit their structures from their parents' (program inheritance). The concept of data inheritance and program inheritance provides an interesting property that can be naturally integrated into the Bayesian inference framework. More theoretical treatment of the IDI method with applications to classification and diagnostic problems can be found in [15]. In this paper, we focus on the speed-up effect of IDI in genetic programming applied to time-series prediction.

The paper is organized as follows. Section 2 describes IDI in more detail. Section 3 presents the experimental results, analyzes the characteristics of evolved data sets, and compares its performance with those of conventional methods. Sections 4 and 5 provide theoretical foundations of the presented method and draws conclusions.

2 Incremental Data Inheritance

2.1 Algorithm Description

The basic idea in genetic programming with incremental data inheritance is that programs and their data are evolved at the same time. With each program is associated a separate data set. Programs are tree-like structures while data are fitness cases used for evolving programs. Thus, IDI distinguishes two populations: program population and data population. The program population $A(g)$ consists of program individuals $A_i(g)$ and the data population $D(g)$ consists of data individuals $D_i(g)$, where g denotes the generation number.

The program population has a fixed size of M, while the size of the data population increases incrementally. The initial program population $A(0)$ is created randomly. The initial data population $D(0)$ of size n_0 is created by fitness cases chosen (at random) from the given basis data set $D^{(N)}$ of size N. Then the data population evolves as the program population evolves. The entire process is summarized in Figure 1.

1. Initialize program population $A(0)$ of size M.
2. Initialize data population $D(0)$ of size n_0. Each data set $D_i(0)$ is randomly sampled from the basis data set $D^{(N)}$ of size N which is given.
3. Set generation count $g \leftarrow 1$.
4. While $(g \leq g_{max})$ do
   (a) Evaluate fitness $F_i(g)$ of programs $A_i(g)$ using associated data sets $D_i(g)$.
   (b) Repeat until (M offspring programs are generated)
      i. (Program inheritance) Select and recombine two parent programs $A_i(g)$ and $A_j(g)$ to generate two offspring programs $A_i(g+1)$ and $A_j(g+1)$.
      ii. (Data inheritance) Recombine two parent data sets $D_i(g)$ and $D_j(g)$ to generate two offspring data sets $D_i(g+1)$ and $D_j(g+1)$ with $|D_i(g+1)| \geq |D_i(g)|$ (see text for more details).
   (c) $g \leftarrow g+1$.

Figure 1: Outline of genetic programming with incremental data inheritance (IDI).

At each generation g, the fitness value $F_i(g)$ of all programs $A_i(g)$ are evaluated using the associated data sets $D_i(g)$. Fitter programs are chosen into the mating pool $B(g)$ and then the mating process is repeated until M offspring programs are produced.

The mating process is divided into two phases: program inheritance and data inheritance (Figure 2). The program inheritance phase evolves child programs, $A_i(g+1)$ and $A_j(g+1)$, from parent programs, $A_i(g)$ and $A_j(g)$, using crossover and mutation. This is the same as in standard genetic programming. The data inheritance phase is similar to the program inheritance phase except it evolves data sets rather than programs.

2.2 Uniform Data Crossover

Several data inheritance mechanisms are possible. We propose a variant of uniform crossover that we call uniform data crossover. A simplified example for illustrating this process is given in Figure 3.

Two parent data sets, $D_i(g)$ and $D_j(g)$, are crossed to inherit their subsets to two offspring data sets, $D_i(g+$

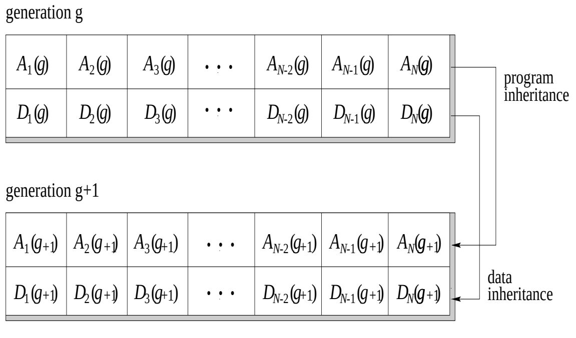

Figure 2: The program inheritance and data inheritance in genetic programming with incremental data inheritance.

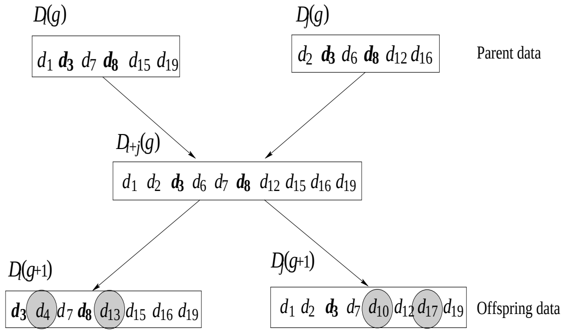

Figure 3: Data crossover: a simplified example.

1) and $D_j(g+1)$. In the uniform data crossover, the data of parents' are mixed into a union set

$$D_{i+j}(g) \quad = \quad D_i(g) \cup D_j(g), \tag{1}$$

which are then redistributed to two offspring $D_i(g+1)$ and $D_j(g+1)$, where the size of offspring data sets are equal to $n_{g+1} = n_g + \lambda$, where $\lambda \geq 1$ is the data increment size. Thus, the size of data sets monotonically increases as generation goes on.

To ensure performance improvement, it is important to maintain the diversity of the training data during evolution. The diversity of data set $D_i(g)$ is measured by the ratio of distinctive examples:

$$d_i = \frac{|D_{i+j}(g)|}{|D_i(g)|} - 1, \qquad 0 \leq d_i \leq 1 \tag{2}$$

where $d_i = 0$ if the parents have the same data and $d_i = 1$ if parents have no common training examples. To maintain the diversity, a portion ρ of the diversity factor d_i is used to import examples from the basis data set.

$$r_i = \rho \cdot (1 - d_i), \qquad 0 \leq \rho \leq 1. \tag{3}$$

This injection can be regarded as a data mutation.

Figure 3 illustrates the process, where two parent data sets of size 6 each are unioned to form the genetic pool of size 10, from which two offspring data sets of size 8 each are inherited. Marked are the data imported from the basis data set to maintain the diversity.

2.3 The Adaptive Fitness Function

Note that each program uses a different set of fitness cases, and thus the fitness function changes as generation goes on. To define our fitness function, let $L(D_i(g)|A_i(g))$ denote a code length for describing data $D_i(g)$ using program $A_i(g)$. We also consider the description length of the program itself, denoted $L(A_i(g))$. The fitness is then defined as their sum:

$$F_i(g) \quad = \quad L(D_i(g)|A_i(g)) + L(A_i(g)). \tag{4}$$

Minimizing this quantity is known as the minimum description length (MDL) principle [8]. The MDL principle leads to finding the most compact program having good predictive performance for unseen data.

The exact calculation of information-theoretic code lengths requires the true probability distribution of underlying data structure which is in most real situations unknown. Instead, we define an adaptive fitness function in its most general form as

$$\begin{aligned} F_i(g) &= L(D_i(g)|A_i(g)) + L(A_i(g)) \\ &= \beta F_D + \alpha F_A \\ &= E(D_i(g)|A_i(g)) + \alpha(g) C(A_i(g)), \end{aligned} \tag{5}$$

where $E(D_i(g)|A_i(g))$ and $C(A_i(g))$ are the measures for error and complexity of the program. Here, we have used the relation $L(D_i(g)|A_i(g)) = -\log P(D_i(g)|A_i(g)) \propto E(D_i(g)|A_i(g))$ with the assumption of Gaussian noise in the training data. The parameter $\alpha(g)$ balances the two factors ($E(\cdot)$ and $C(\cdot)$) as follows

$$\alpha(g) = \begin{cases} \frac{1}{n_g^2} \frac{E_{best}(g-1)}{\hat{C}_{best}(g)} & \text{if } E_{best}(g-1) > \epsilon \\ \frac{1}{n_g^2} \frac{1}{E_{best}(g-1) \cdot \hat{C}_{best}(g)} & \text{otherwise.} \end{cases} \tag{6}$$

This is the adaptive Occam method [16] in which n_g is now a variable that increases monotonically as a function of generation g. User-defined constant ϵ specifies the maximum training error allowed for the run. $E_{best}(g-1)$ is the error of the best model of generation $g-1$. $\hat{C}_{best}(g)$ is the size of the best model at generation g estimated at generation $g-1$. These are used to balance the error and complexity terms to obtain models as parsimonious as possible while not sacrificing their accuracy.

In the experiments, we use the following error measure:

$$E(D_i(g)|A_i(g)) = \frac{1}{n_g} \sum_{(x_c,y_c)\in D_i(g)} (f_i(x_c) - y_c)^2, \quad (7)$$

where $f_i(x_c) = f(x_c; A_i(g))$ is the output of the programs $A_i(g)$ for input x_c and n_g is the size of the training set $D_i(g)$. The program complexity can be defined in various ways since the adaptive Occam method uses relative complexity rather than absolute values. Typically, the complexity of genetic trees is defined as the number of nodes in the tree and other additive terms.

3 Application to Time Series Prediction

3.1 The Problem and the Method

Figure 4 shows a series of 2000 measurements of chaotic intensity fluctuations. This data was generated from far-infrared NH_3 laser in a physics laboratory [6]. This problem was used as a benchmark in the 1992 Santa Fe time series competition. We used the first 1000 data points for evolving the programs and the rest 1000 data points for testing the generalization performance. The training data was generated from the time series as follows: three contiguous values $x_1(t)$, $x_2(t)$, $x_3(t)$ were used as input for the tth training pattern, and the immediate next point $x_4(t)$ was used as the target value $y(t)$ to be predicted. The input attributes of all data sets were linearly scaled into the interval $[0, 1]$. The output attribute has continuous values between 0 and 1.

We used neural trees for program structures. A neural tree consists of nonterminal nodes and terminal nodes [16]. The nonterminal nodes represent neural units and the neuron type is an element of the basis function set $\mathcal{F} = \{\text{neuron types}\}$. Each terminal node is labeled with an element from the terminal set $\mathcal{T} = \{x_1, x_2, ..., x_n\}$, where x_i is the ith component of the external input $\mathbf{x}$. Each link (j, i) represents a directed connection from node j to node i and is associated with a value w_{ij}, called the synaptic weight.

The root node is also called the output unit and the terminal nodes are called input units. Nodes that are neither input nor output units are hidden units. The layer of a node is defined as the longest path length to any terminal node of its subtrees.

Different neuron types are distinguished in the way of computing net inputs. Sigma units compute the sum of weighted inputs from the lower layer:

$$net_i = \sum_j w_{ij} y_j \quad (8)$$

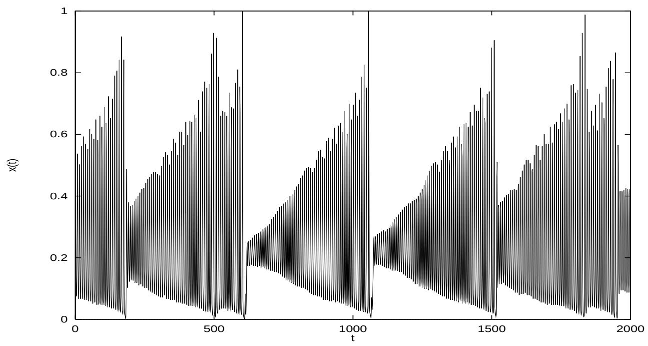

Figure 4: Laser intensity time series used for training ($t = 1, ..., 1000$) and testing ($t = 1001, ..., 2000$) of neural tree programs.

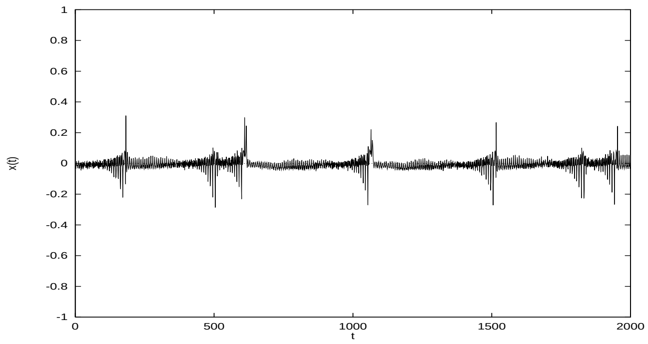

Figure 5: Laser intensity time series: prediction error.

where y_j are the inputs to the ith neuron. Pi units compute the product of weighted inputs from the lower layer:

$$net_i = \prod_j w_{ij} y_j \quad (9)$$

where y_j are the inputs to i.

The output of a neuron is computed either by the threshold response function

$$y_i = \sigma(net_i) = \begin{cases} 1 & : \ net_i \geq 0 \\ -1 & : \ net_i < 0 \end{cases} \quad (10)$$

or the sigmoid transfer function

$$y_i = f(net_i) = \frac{1}{1 + e^{-net_i}} \quad (11)$$

where net_i is the net input to the unit computed by equations (8) or (9).

3.2 Results

Figure 5 plots the difference between the true values and one-step-ahead prediction values for the 1000

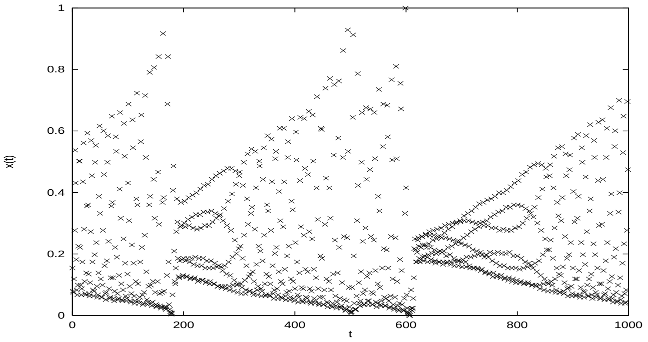

Figure 6: All the training data points given for the laser time series.

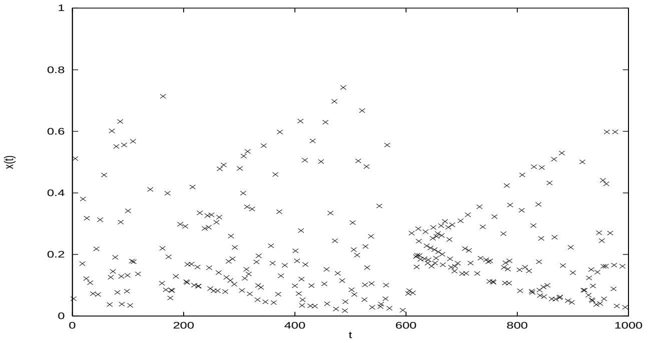

Figure 7: The data points selected by the best neural tree at $g = 30$, using GP with incremental data inheritance.

training points and the following 1000 test points. It can be seen that the neural tree evolved by GP with incremental data inheritance has captured the general trend of the time series. The regions where larger prediction errors were occurred are the turning points of the time series which are known to be difficult to predict. Note that the results have relatively good generalization performance.

In the following we want to analyze the data selection behavior of the incremental data inheritance method. Figure 6 shows the 1000 data points for the entire training set that are given. From these data, the IDI method started genetic programming with an initial data set of size 40 which was increased by 8 each generation. To see the evolution of data sets we depicted in Figure 7 the data points of the best individual at generations $g = 30$. Comparison of Figures 6 and 7 indicates that the sampled subset of the original data set is representative.

To further investigate the efficiency of the incremental data inheritance method, we compared its performance with two other methods. One is standard genetic programming in which the baseline data set

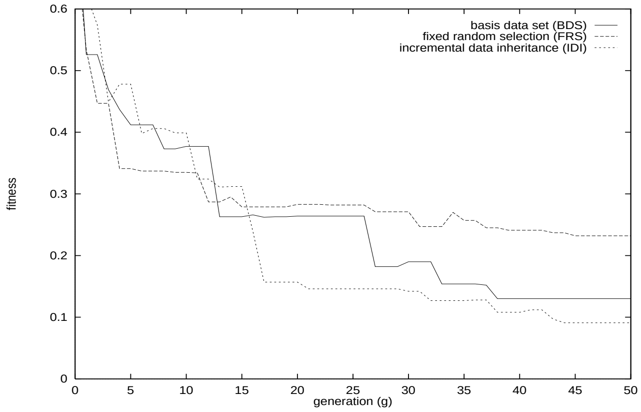

Figure 8: Fitness vs. generation for laser intensity time series.

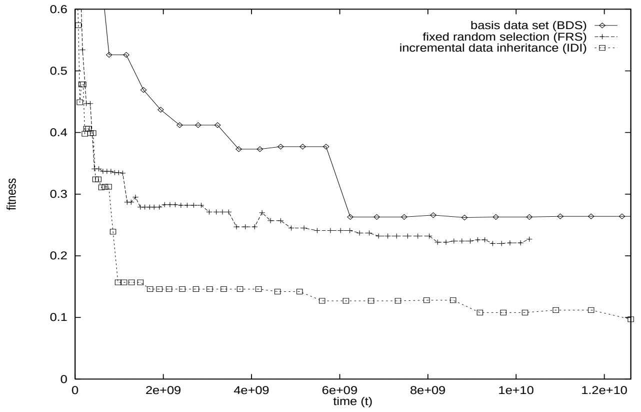

Figure 9: Fitness vs. pseudo-cpu time for laser intensity time series.

(BDS) of size 1000 is used. The second is the method of fixed random selection (FRS) in which a fixed size (in this case 200) of data sets chosen at random for each generation was used.

Figure 8 compares the fitness evolution for the various data selection methods as a function of generation. The incremental selection method (IDI) converged faster than BDS and FRS.

To see the real speed-up, we measured the computational time as a product of the population size and the data size for each generation. The time for data selection can be ignored since most of the execution time is spent on fitness evaluation, especially for fine-tuning of weights. The performance in this pseudo-cpu time is compared in Figure 9. The speed-up effect of IDI is even clearer here. The performance of IDI improved very fast, especially in the early generations, and approached a fitness level of approximately 0.1. Fixed random selection also improved its performance relatively fast, but its fitness stocked around 0.25. The baseline method (BDS) was the slowest and its fitness converged to around 0.27 when IDI reached 0.1. Though BDS could eventually reach the performance level of IDI if run further, but it took so long that this cannot be seen in Figure 9. In effect, the genetic programming with incremental data inheritance achieved for this problem a speed-up of up to an order of magnitude compared to the baseline algorithm.

4 Discussion

4.1 Four Key Properties of IDI

The incremental data inheritance method can be characterized by the four properties: separate data sets for programs, same data size for each generation, data inheritance, and monotonic growth of data size during evolution.

1. *Separate data sets.* Each model $A_i(g)$ in the population has its own data $D_i(g)$, where $D_i(g)$ is a subset of the original data set $D^{(N)}$ of size N.

2. *Same data size.* The sizes of data sets are equal for all programs at the same generation, but their elements may be different:
$$|D_i(g)| = |D_j(g)|, \; D_i(g) \neq D_j(g), \quad i, j = 1, ..., M. \tag{12}$$

3. *Data inheritance.* Just as programs inherit their structural features from their parents, data sets $D_i(g)$ of programs inherit partial data from their parents' $D_p(g-1)$ and $D_q(g-1)$, where p and q are parents of i.

4. *Monotonic growth.* Data sets grow monotonically as generation goes on, i.e. $|D_i(g)| > |D_i(g-1)|$.

All these features together constitute the incremental data inheritance method. The monotonicity property is important to ensure the method ultimately uses all the given data. This guarantees the method not to loose known information about the problem domain. This is contrasted with DSS [5] where only one common data set for each generation exists, data are chosen randomly, and the data size remains fixed through the generations. This is also contrasted with RAT where the data size arbitrarily changes from generation to generation.

The experimental results support the importance of both "incremental selection" and "data inheritance" concepts adopted in the IDI algorithm. Without the "incremental" component, there is no guarantee for reaching the desired fitness level, as shown in the case of fixed random selection. Using data inheritance rather than random sampling, the induction process is accelerated. This is because by keeping separate data for each program, the training can focus on new data rather than the whole data.

4.2 Bayesian Aspects of GP with IDI

The genetic programming with incremental data inheritance can be formulated as attempting to find the best program using as small a training set as possible, where the time limit is given as the maximum generation g_{max}. The goodness of the programs are defined as the fitness function $F_i(g)$. Thus, we can formulate the GP with IDI as:

$$A_{best}^{(g)} = \min_{g \leq g_{max}} \mathrm{argmin}_{A_i(g), D_i(g)} \; F_i(g) \tag{13}$$

where $A_i(g) \in \mathcal{A}$ and $D_i(g) \subset D^{(N)}$, where $\mathcal{A}$ is the space of all possible programs considered. It should be noted that, in contrast to conventional GP, here the data sets $D_i(g)$ as well as the programs $A_i(g)$ are evolved.

The process (13) can be interpreted as a Bayesian inference where the maximum a posteriori (MAP) program is sought. To see this, let $P_i(g)$ denote the posterior probability of programs, i.e. the probability of the program $A_i(g)$ given the data $D_i(g)$:

$$P_i(g) \equiv P(A_i(g)|D_i(g)). \tag{14}$$

Using Bayes rule, this can be written as

$$P_i(g) = \frac{P(D_i(g)|A_i(g))P(A_i(g))}{\sum_{A_j(g), D_j(g)} P(D_j(g)|A_j(g))P(A_j(g))}. \tag{15}$$

Now, we define the fitness function as the minus logarithm of $P_i(g)$:

$$F_i(g) \equiv -\log(P_i(g)) \tag{16}$$

which should be minimized. Substituting this into Eq. (13) and using Bayes rule (15), we get

$$\begin{aligned} A_{best}^{(g)} &= \min_{g \leq g_{max}} \mathrm{argmin}_{A_i(g), D_i(g)} \; F_i(g), \\ \min F_i(g) &= \min\{-\log P(D_i(g)|A_i(g)) \\ &\quad - \log P(A_i(g))\} \qquad (17) \\ &= \min\{L(D_i(g)|A_i(g)) + L(A_i(g))\}, \end{aligned}$$

which leads to our MDL-based fitness function (5).

This shows that our formulation of genetic programming can be regarded as a Bayesian inference. In other words, the ideal Bayesian inference is approximated by a more practical evolutionary search method based on genetic programming, where the space of programs to be searched is limited to the size of program population and the allowed computing time is limited to the maximum generation number. The role of incremental data inheritance in this process is to reduce the time for estimating the probability of programs. Our experimental analysis shows this can be done without loss of predictive accuracy of the programs.

5 Conclusions

We have presented a method for accelerating evolution speed of genetic programming by incrementally evolving subsets of given fitness cases. Since the fitness evaluation step is a bottleneck in GP computing time, this method can make an essential contribution to improving the GP performance. In contrast to fixed-size subset selection method, such as dynamic subset selection, the incremental selection method has the advantage that the effective data size is determined automatically during evolution.

Experimental results have shown that by reducing the fitness cases the evolution speed of GP can be enhanced without loss of generality of the evolved programs. This is especially true for problem settings in which a large amount of fitness cases are available. In this case, the active data inheritance can exploit the redundancy in the data, while the standard GP blindly re-evaluates all the fitness cases.

Genetic programming with incremental data inheritance (IDI) can be interpreted as co-evolution of programs and their data sets. While existing GP methods evolve only the programs, the GP with incremental data inheritance evolves data sets associated with the programs. Since the evolution of programs depends on data sets and the evolution of data sets depends on programs, both the program and its data concurrently evolve towards an optimal combination. This can be viewed as a genetic programming version of the genetic-neural co-evolutionary learning process in GENIE [12]. From the efficiency point of view, it is interesting that IDI results in accelerated evolution of programs through minimal use of fitness cases.

Acknowledgments

This research was supported in part by the Korea Science and Engineering Foundation (KOSEF) under grants 96-0102-13-01-3 and 981-0920-350-2.

References

[1] P. J. Angeline, and J. B. Pollack (1993). Competitive environments evolve better solutions for complex tasks. In *Proc. 5th Int. Conf. on Genetic Algorithms*, 264-270. Morgan Kaufmann.

[2] W. Banzhaf, P. Nordin, R. Keller, and F. Francone (1998). *Genetic Programming - An Introduction*, Morgan Kaufmann, San Francisco, CA.

[3] J. M. Daida, F. Bersano-Begey, S. J. Ross, and J. F. Vesecky (1996). Computer-assisted design of image classification algorithms: Dynamic and static fitness evaluations in a scaffolded genetic programming environment. In J.R. Koza (eds.). *Genetic Programming 1996*, 279-284. Cambridge, MA: The MIT Press.

[4] C. Gathercole, and P. Ross (1994). Dynamic training subset selection for supervised learning in genetic programming. In Y. Davidor, H.-P. Schwefel, and R. Männer, (eds.). *Parallel Problem Solving from Nature III*, 312-321. Berlin: Springer-Verlag.

[5] C. Gathercole, and P. Ross (1997). Small populations over many generations can beat large populations over few generations in genetic programming. In J.R. Koza (eds.). *Genetic Programming 1997*, 111-118. Cambridge, MA: The MIT Press.

[6] H. Hübner, C. O. Weiss, N. B. Abraham, and D. Tang (1993). Lorenz-like chaos in nh_3-fir laser. In Weigend, A. and Gershenfeld, N., editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, 73-104. Addison-Wesley.

[7] John R. Koza (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.

[8] J. Rissanen (1986). Stochastic complexity and modeling. *The Annals of Statistics*, 14:1080-1100.

[9] E. V. Siegel (1994). Competitively evolving decision trees against fixed training cases for natural language processing. In K. E. Kinnear, Jr., (Ed.), *Advances in Genetic Programming*, Chapter 19. Cambridge, MA: The MIT Press.

[10] T. Soule, J. A. Foster, and J. Dickinson (1996). Code growth in genetic programming. In J.R. Koza (eds.). *Genetic Programming 1996*, 215-223. Cambridge, MA: The MIT Press.

[11] A. Teller, and D. Andre (1997). Automatically choosing the number of fitness cases: The rational allocation of trials. In J.R. Koza (eds.). *Genetic Programming 1997*, 321-328. Cambridge, MA: The MIT Press.

[12] B.-T. Zhang (1992). *Learning by Genetic Neural Evolution*, (in German), DISKI Vol. 16, 268 pages, ISBN 3-929037-16-6, Infix-Verlag, St. Augustin/Bonn.

[13] B.-T. Zhang (1993). Self-development learning: Constructing optimal size neural networks via incremental data selection, *Arbeitspapiere der GMD* 768, German National Research Center for Information Technology (GMD), St. Augustin/Bonn, July 1993.

[14] B.-T. Zhang (1994). Accelerated learning by active example selection, *International Journal of Neural Systems*, 5(1): 67-75.

[15] B.-T. Zhang (1999). A Bayesian framework for evolutionary computation, *Proc. 1999 Congress on Evolutionary Computation* (CEC99), Special Session on Theory and Foundations of Evolutionary Computation, Washington, D.C., IEEE Press, 1999 (to appear).

[16] B.-T. Zhang, P. Ohm, and H. Mühlenbein (1997). Evolutionary induction of sparse neural trees. *Evolutionary Computation*, 5(2): 213-236.

[17] B.-T. Zhang, and G. Veenker (1991). Focused incremental learning for improved generalization with reduced training sets, *Proc. Int. Conf. Artificial Neural Networks*, ICANN-91, Kohonen, T. *et al.* (eds.) North-Holland, 227-232.

[18] B.-T. Zhang, and G. Veenker (1991). Neural networks that teach themselves through genetic discovery of novel examples, *Proc. Int. Joint Conf. on Neural Networks*, IJCNN-91, 690-695. IEEE Press.