Enhancing Robustness of Genetic Programming at the Species Level

Byoung-Tak Zhang
Dept. of Computer Engineering
Seoul National University
Seoul 151-742, Korea
btzhang@comp.snu.ac.kr

Je-Gun Joung
Dept. of Computer Engineering
Konkuk University
Seoul 143-701, Korea
jgjoung@pluto.konkuk.ac.kr

ABSTRACT

Many studies on genetic programming have focused on improving robustness of evolved programs at the individual level. The applicability of this approach is limited since the evolution of perfect solutions is difficult or very expensive when the fitness cases are sparse or noisy. In this paper we present an alternative method that attempts to enhance robustness of genetic programming at the species level. Each genetic program in the population is regarded as a cooperating expert having a voting factor in addition to the usual fitness value. The voting factors are learned from the training set and used to make the final decision from the decisions of the best *n* experts evolved by genetic programming. Experimental results are provided that show combination of the decisions of multiple genetic programs can outperform the predictive accuracy of the best fit program alone.

1 Introduction

One of the basic characteristics of genetic programming [Koza, 1992] is that it discovers programs through a parallel search on the basis of a population, instead of a single individual. As a byproduct of population-based search, one gets a number of potentially effective solutions. However, most genetic programming processes choose just a single best member of the population as their final solution and the rest of the population are discarded. By doing this, the computation time and memory space used for the evolution of the entire population are thrown away.

Many studies on genetic programming have focused on robustness of evolved programs at the individual level. For example, Reynolds [Reynolds, 1994] studied the evolution of obstacle avoidance behavior using genetic programming. He observed that even if a genetic program successfully controls the robot for a given task on a specific environment, it does not necessarily show a robust behavior on the other environment. Ito et al. [Ito et al., 1996] have examined the robustness of the genetic programs in the context of the box moving problem [Koza, 1992]. Both Reynolds and Ito et al. report an improvement of robustness by evolving on perturbed fitness cases.

The method proposed in this paper involves collective robustness of a subset of the population evolved by genetic programming, i.e. we address species-level robustness. The presented method can be used in combination with the individual-level robustness enhancement methods.

We regard genetic programs as cooperating experts and the decision is made on the basis of a pool of the best *n* experts from the population, instead of a single expert. The decision making proceeds as follows: The same fitness case is fed to all programs of the pool. Each program makes a prediction and these predictions are grouped together to form the instance that is fed to the master algorithm called MGP (Mixing Genetic Programs). The master algorithm then makes its prediction based on the voting factors assigned to each program. The voting factors of the master algorithm is calculated from the training examples. We have experimented with various weighting schemes, including naive majority voting, additive weighting, and multiplicative weighting [Littlestone and Warmuth, 1994].

The MGP method seems especially useful when the training data is sparse [Frankone et al., 1996] or noisy where the evolution of perfect best individual is almost impossible. The results suggest that a set of imperfect solutions or immature programs can be used to make decisions that can be made by the best program theoretically possible, without explicitly evolving the best program.

The paper is organized as follows. In Section 2 we describe various schemes for learning and using the voting factors. Section 3 describes the genetic programming method and experimental set-up. Section 4 reports experimental results and their analysis. Section 5 contains conclusions and directions for future work.

2 Combining the Decisions of Multiple Genetic Programs

We investigate the situation where we are given a pool of n programs that have varying fitness values. Each program is considered as an expert or a prediction strategy. We propose to use a master algorithm that combines the predictions of the pool to make its own decision.

At the beginning of trial t, the master algorithm feeds the given observation x_t to all experts. The master then uses some function of the n predictions produced by the experts to form its own prediction $\hat{y}_t$. At the end of the trial the feedback y_t is shared with all experts to change the voting weight of each expert. In this setting, all predictions and outcomes are boolean. This is the problem solved by the basic Weighted Majority (WM) algorithm [Littlestone and Warmuth, 1994].

We describe a master algorithm as it applies to finite pools of tree-structured programs evolved by gentic programming. The algorithm is summarized in Figure 1 and shall be referred to as MGP. Let X denote the set of possible observations and $\{0,1\}$ the two possible outcomes. We use

$$D_N = \{(x_t, y_t)\}_{t=1}^{N} \tag{1}$$

for the set of training fitness cases over $(X \times \{0,1\})$. On the tth trial, each expert V_i makes the prediction

$$\tilde{y}_t^{(i)} = f(x_t; V_i), \tag{2}$$

where $x_t \in X$ is the current observation. At the end of the trial the expert is given the feedback $y_t \in \{0,1\}$ for the current trial. We say that expert V_i is wrong, makes a mistake, or is incorrect when its prediction at trial t, $\tilde{y}_t^{(i)}$, is different from y_t.

The MGP algorithm works as follows. Initially a positive weight is associated with each program of the pool. Unless otherwise stated, all weights are initially one. For a given input x_t, algorithm MGP forms its prediction $\hat{y}_t$ by comparing the total weight q_0 of the programs of the pool that predict 0 to the total weight q_1 of the programs predicting 1:

$$\hat{y}_t = \text{Voting}(x_t, \tilde{y}_t^{(1)}, ..., \tilde{y}_t^{(n)}) \tag{3}$$

where $\tilde{y}_t^{(i)}$ denotes the output of the ith expert for the tth trial.

More specifically, let S_0 and S_1 denote the subset of programs of the pool that predict 0 and 1, respectively:

$$S_0 = \{\, i \mid f(x_t; V_i) = 0,\ i = 1, ..., n\}, \tag{4}$$
$$S_1 = \{\, i \mid f(x_t; V_i) = 1,\ i = 1, ..., n\}. \tag{5}$$

Then the total weights, q_0 and q_1, of the programs in each subset are computed

$$q_0 = \sum_{i \in S_0} v_i, \tag{6}$$
$$q_1 = \sum_{i \in S_1} v_i. \tag{7}$$

MGP predicts according to the larger total (arbitrarily in case of a tie):

$$\begin{aligned} &\text{If } q_1 > q_0, \text{ then } \hat{y}_t = 1,\\ &\text{else if } q_1 < q_0, \text{ then } \hat{y}_t = 0,\\ &\text{else assign } \hat{y}_t \text{ a 1 or 0 randomly.}\end{aligned} \tag{8}$$

When MGP makes a mistake, the weights of those experts of the pool that disagreed with the target output are adjusted. It should be noticed that the weights are changed only when the master algorithm, not the individual expert, predicts incorrectly. Thus it is possible that an expert actually made a mistake for x_t, but its voting weight is not adjusted.

The learning component of the MGP algorithm takes the training set as input to produce as output the weight for voting:

$$v_i = \text{Learning}(x_t, y_t, \tilde{y}_t^{(i)}, \hat{y}_t) \tag{9}$$

where $\tilde{y}_t^{(i)}$ and $\hat{y}_t$ denotes the output of the ith expert and of the master algorithm for the tth trial. We have studied several strategies for assigning the voting factors, including naive voting, additive weighting, and multiplicative weighting.

In naive voting, all the experts have the voting factor of constant value γ:

$$v_i \leftarrow \gamma, \tag{10}$$

where v_i is the voting weight of ith expert for the tth trial. In this scheme the voting factor v_i remains constant and no learning takes place.

In multiplicative weighting, when MGP makes a mistake, the weights of those experts of the pool that disagreed with the outcome are each multiplied by a fixed γ, i.e. the voting weight is adapted by the following rule:

$$v_i \leftarrow v_i \times \gamma \tag{11}$$

where γ is a constant in $[0,1)$. This scheme assigns to each expert a weight of the form γ^{m_i}, where

```
Procedure MixingGeneticPrograms($n$, $\sigma$, $\gamma$)
  $n$: number of experts
  $\sigma$: weighting strategy
  $\gamma$: weighting factor
STEP 1. Divide the known data into two distinct sets:
  training set $D$ and test set $C$.
STEP 2. Use genetic programming to evolve
  a population $\mathcal{A}$ of $N_{pop}$ genetic programs.
STEP 3. Choose a pool $\mathcal{V}$ of $n$ experts from $\mathcal{A}$.
STEP 4. Set the voting factors $v_i = 1.0$, for i = 1, ..., n.
STEP 5. //Learning the voting factors of experts //
  For each training example $(x_t, y_t) \in D$, $t = 1, ..., N$
    Set $q_1 = 0.0$ and $q_0 = 0.0$.
    For each voting program $V_i \in \mathcal{V}$, $i = 1, ..., n$
        Present $x_t$ to $V_i$ to produce $\tilde{y}_t^{(i)} = f(x_t; V_i)$.
        If $\tilde{y}_t^{(i)} = 1$, then $q_1 = q_1 + v_i$, else $q_0 = q_0 + v_i$.
        If $\tilde{y}_t^{(i)} \neq y_t$, then MISTAKE($i$) = TRUE,
            else MISTAKE($i$) = FALSE.
    If $q_1 > q_0$, then $\hat{y}_t = 1$,
    else if $q_1 < q_0$, then $\hat{y}_t = 0$,
    else assign $\hat{y}_t$ a 1 or 0 at random.
    If $\hat{y}_t \neq y_t$, then for $i = 1, ..., n$
        If MISTAKE($i$), then
            If $\sigma$ = naive then $v_i \leftarrow \gamma$,
            else if $\sigma$ = additive then $v_i \leftarrow v_i - \gamma$,
            else $\sigma$ = multiplicative then $v_i \leftarrow v_i \times \gamma$.
STEP 6. // Estimating the predictive accuracy //
  For each test example $(x_t, y_t) \in C$, $t = 1, ..., N_{test}$
    Set $q_1 = 0.0$ and $q_0 = 0.0$.
    For each voting program $V_i \in \mathcal{V}$, $i = 1, ..., n$
        Present $x_t$ to $V_i$ to produce $\tilde{y}_t^{(i)} = f(x_t; V_i)$.
        If $\tilde{y}_t^{(i)} = 1$, then $q_1 = q_1 + v_i$,
        else $q_0 = q_0 + v_i$.
    If $q_1 > q_0$, then $\hat{y}_t = 1$,
    else if $q_1 < q_0$, then $\hat{y}_t = 0$,
    else assign $\hat{y}_t$ a 1 or 0 randomly.
    If $\hat{y}_t \neq y_t$, then $m = m + 1$.
```

Figure 1: Summary of the MGP algorithm.

m_i is the total number of mistakes incurred by the ith expert. The essential property is that the experts making many mistkes get their weights rapidly slashed. This strategy is adopted by the WM algorithm [Littlestone and Warmuth, 1994].

Additive weighting is similar to multiplicative weighting. As in multiplicative voting, additive voting assigns to each expert a weight proportional to the number of mistakes m_i incurred by the ith expert. But it differs from multiplicative weighting in that the functionality is addition instead of multiplication, i.e. for each mistake incurred by the ith expert, the additive scheme adjusts its weight as follows:

$$v_i \leftarrow v_i - \gamma \qquad (12)$$

where γ is a constant in $[0, 1)$. Typically $\gamma = \frac{1}{N}$, where N is the total number of training fitness cases. The additive scheme assigns to each expert a weight of the form $m_i\gamma$, where m_i is the total number of mistakes incurred by the ith expert. This type of weighting produces a smooth weight landscape, giving less fit expert a fair chance of influence.

3 Experiments

We compared the performance of the best expert to that of the mixture of multple experts. The problems used for experiments were multiplexer problems. The task of a boolean multiplexer is to decode a binary address register (A0, A1, ...) and return the value of the data register (D0, D1, ...) at the address. We studied two k-multiplexer problems with $k = 6, 11$.

The goal of experiments with the 6-multiplexer problem was to study the behavior of the MGP algorithm when genetic programming is based on noisy training data. The training set contained 64 examples with 20% of their output corrupted. In contrast, we used a set of 64 correct examples, the set of all possible examples, for the test of generalization performance. By doing this, the sampling bias for generating the test set can be eliminated and still the predictive power of several MGP strategies can be objectively compared. A side effect of this experimental setting is that in our experiments the predictive accuracy of genetic programs can be better than the training accuracy, which is unusual in many real-life situations.

The second set of experiments was performed for the situation in which training data is clear but sparse. This case is studied on the 11-multiplexer problem for which the evolution of a perfect solution by genetic programming with a small population for a small number of generations was found to be nontrivial. For this problem we used a training set of size 512, i.e. 25% of the en-

Design Parameters	*Values Used*
task	6-multiplexer
terminal set	{ A0, A1, D0, D1, D2, D3 }
function set	{ IF, AND, OR, NOT }
population size	200
max generation	200
crossover rate	0.9
mutation rate	0.1
training set	64 examples with 20% noise
test set	64 correct examples
number of experts	n = 1, 5, 11, 21, 51, or 101
weighting strategies	naive/additive/multiplicative
γ values	$\gamma_n = 1$, $\gamma_a = \frac{1}{64}$, $\gamma_m = 0.9$

Table 1: Experimental set-up for the 6-multiplexer problem.

Design Parameters	Values Used
task	11-multiplexer
terminal set	{ A0, A1, A2, D0, D1, ..., D7 }
function set	{ IF, AND, OR, NOT }
population size	500
max generation	200
crossover rate	0.9
mutation rate	0.1
training set	512 correct random examples
test set	2048 correct examples
number of experts	n = 1, 5, 11, 21, 51, or 101
weighting strategies	naive/additive/multiplicative
γ values	$\gamma_n = 1$, $\gamma_a = \frac{1}{512}$, $\gamma_m = 0.9$

Table 2: Experimental set-up for the 11-multiplexer problem.

tire example space. The test set contained 2048 correct examples.

The function set for solving the multiplexer problems contained IF, AND, OR, NOT which take 3, 2, 2, 1 arguments respectively. Tables 1 and 2 summarize the design parameters for the experiments. The crossover operation exchanges substrees of two individuals chosen at random. Mutation consists of changing AND to OR and vice versa. Mutation also changes the labels of terminal nodes.

Fitness is measured by a simplified version of the adaptive Occam method [Zhang and Mühlenbein, 1996]:

$$F_i(g) \quad = \quad E_i(g) + \alpha(g) C_i(g), \tag{13}$$

where $E_i(g)$ is the sum of total errors of ith program at generation g for the training set and $C_i(g)$ is the complexity penalty measured as the sum of the number of nodes and the depth of the program tree. The Occam factor was:

$$\alpha(g) = \frac{1}{N^2} \frac{E_{best}(g-1)}{C_{best}(g-1)} \tag{14}$$

where N is the size of training set.

We compared the performances of three MGP strategies described in the last section. The γ values chosen were $\gamma_n = 1$ (for naive voting), $\gamma_a = \frac{1}{N}$ (for additive weighting), $\gamma_m = 0.9$ (for multiplicative weighting). All the experts were trained using the same set of training data. The test data was the same for all MGP strategies.

4 Results

Figure 2 shows the average performance of various decision strategies for the noisy 6-multiplexer problem. All the results are average values for 50 runs. To analyze the behavior of MGP, we applied MGP after every generation of genetic programming. This is not necessary; normally MGP will be applied only after the evolution is complete. The F_{best} durve shows the training error and the G_{best} curve plots the generalization error of the best fit program. The nMGP(5) and mMGP(5) curves show the generalization performances obtained by the naive MGP and the multiplicatively weighted MGP strategies, respectively. The number of experts was five.

This result clearly demonstrates that making decisions by combining multiple programs can lead to better predictions than using simply the best program evolved. Figure 3 confirms this by showing the differences

$$G_{best} - \text{nMGP}(5) \quad \text{and} \quad G_{best} - \text{mMGP}(5). \tag{15}$$

We also compared the performance of naive MGP and multiplicative MGP. As can be seen in Figure 4 which plots the difference

$$\text{nMGP}(5) - \text{mMGP}(5). \tag{16}$$

The results indicate that the prediction accuracy of weighted majority is consistently superior to the naive majority strategy.

We also investigated the effect of the varying numbers of experts on the predictive performance for the weighted mixing strategy. Shown in Figures 5 – 8 are the results for $n = 11, 21, 51, 101$. All results are average values derived from 50 runs. Each figure also shows the training and generalization performance of the best fit program for comparison purposes. Comparing the figures shows the tendency that combining multiple programs increases the performance of the best individual with a possible risk of degenerating the robustness when too many programs take part in decision making. The

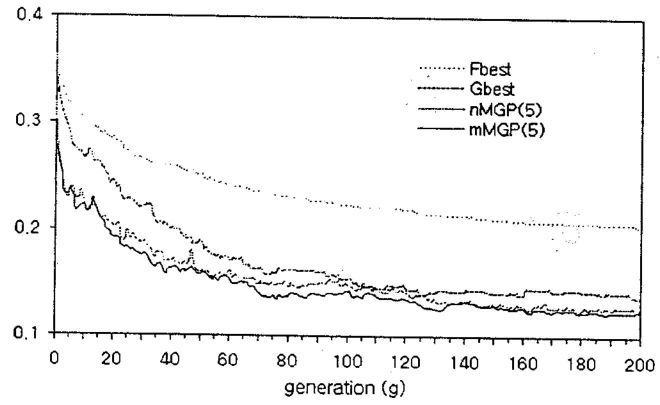

Figure 2: Average performance of naive voting (nMGP) vs. multiplicatively weighted voting (mMGP) strategies for the noisy 6-multiplexer problem ($n = 5$).

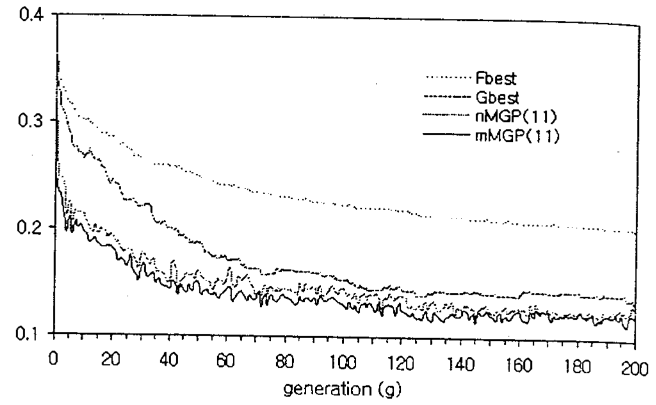

Figure 5: Performance comparison for varying numbers of experts for the weighted mixing genetic programming ($n = 11$).

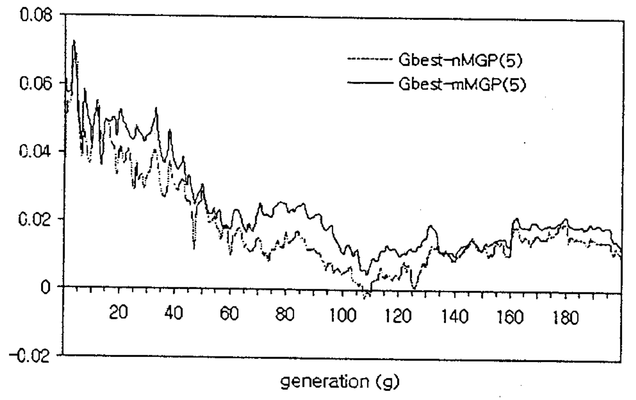

Figure 3: Comparison of predictive performance of the single best program against two different mixtures of five genetic programs.

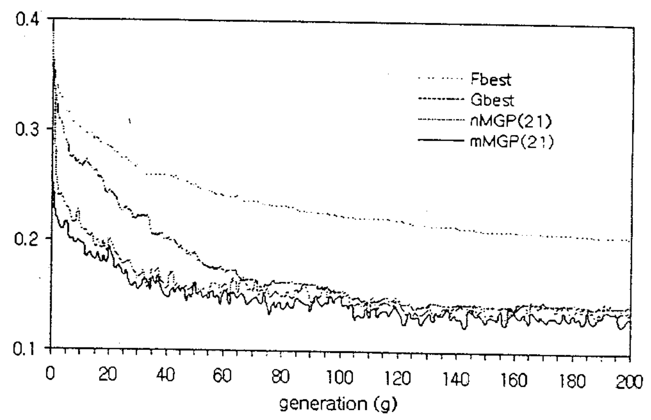

Figure 6: Performance comparison for varying numbers of experts for the weighted mixing genetic programming ($n = 21$).

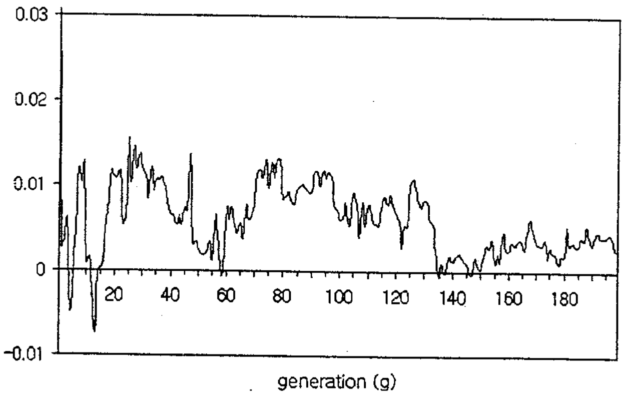

Figure 4: Comparison of predictive performance of the two different mixing strategies: naive mixing (nMGP) and weighted mixing (mMGP).

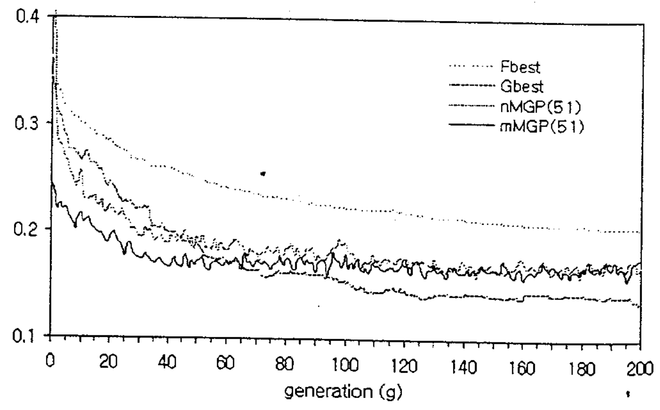

Figure 7: Performance comparison for varying numbers of experts for the weighted mixing genetic programming ($n = 51$).

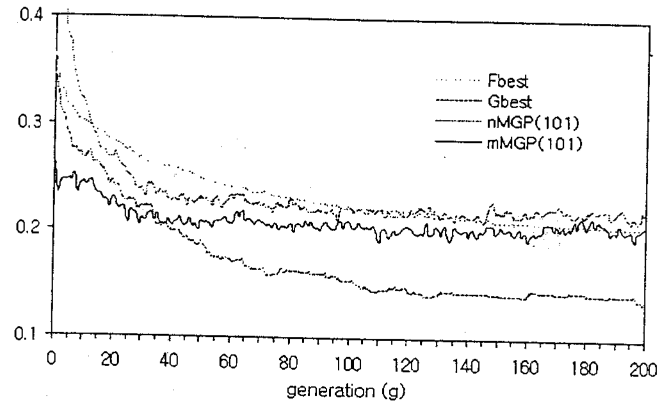

Figure 8: Performance comparison for varying numbers of experts for the weighted mixing strategy ($n = 101$).

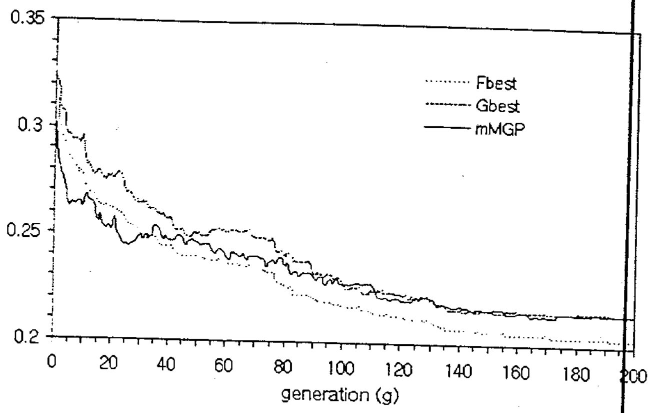

Figure 11: Average performance of naive vs. weighted mixing strategies for the 11-multiplexer problem ($n = 5$).

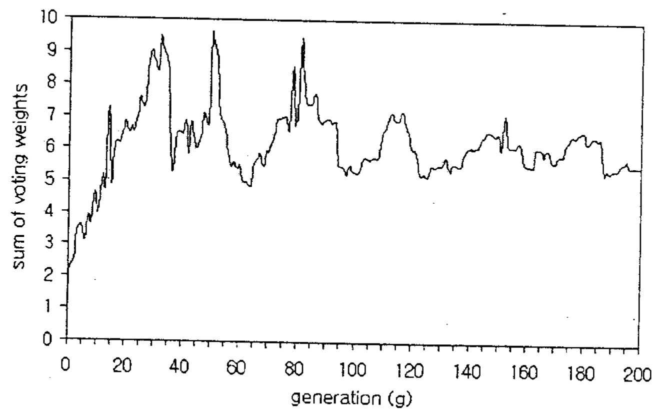

Figure 9: Change of total voting factors of n experts as a function of generations.

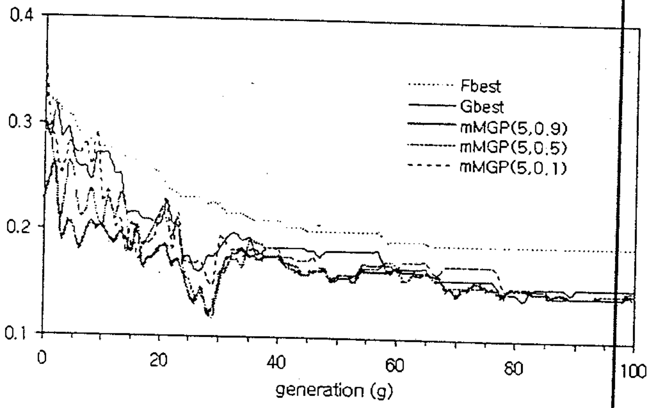

Figure 12: Effect of different weighting factors in multiplicative weighting.

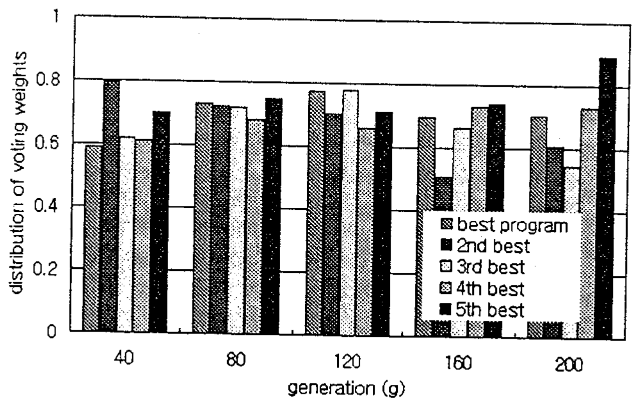

Figure 10: Distribution of voting factors as generation goes on.

latter harmful effect seems caused by the increase of the number of poor programs as the number of experts increases.

To further examine the effect of the number of experts, we analyzed the change of the voting weights and their sum as generation goes on. If we denote the voting weight of the ith best program at generation g by $v_i(g)$, the sum of the voting values denoted $v_{sum}(g)$ is:

$$v_{sum}(g) = \sum_{i=1}^{n} v_i(g), \qquad (17)$$

where $n = 5$ in this experiment.

Figure 9 depicts the change of the total voting weights of a typical run. It can be observed that the $v_{sum}(g)$ value tends to increase as evolution proceeds. This seems attributed to the fact that the best five programs get fitter and make less mistakes as generation goes on. The specific distributions of the values at generations $g = 1, 50, 100, 150, 200$ are shown in Figure 10. As evolution proceeds, the best fit program tends to get higher voting weights than the less fit programs.

Figure 11 shows the results for the 11-multiplexer problem. The results are qualitatively similar to those of Figure 2. The curves are average performance for 10 runs.

5 Conclusions and Future Work

We described a method for combining multiple genetic programs to enhance robustness of genetic programming. Experimental results have shown that combining the decisions of multiple programs can significantly improve the predictive accuracy of the best fit program alone. This species-level approach to improving robustness seems especially useful when training data is sparse or noisy and thus the evolution of perfect best individual is difficult. The results indicate that a set of imperfect solutions or immature programs can be used to make decisions that can be made by the best program theoretically possible, without explicitly evolving the best program.

The MGP algorithm can be used for several purposes. First, the method can be used for speeding-up the genetic programming process. Instead of attempting to evolve a highly fit solution for which g generations are needed, one can use multiple solutions of lower fitness evolved for a small number of generations, say $g/2$, to achieve a comparable prediction accuracy. In this regard, MGP accelerates the evolution speed of genetic programming. Put in another way, MGP relieves genetic programming from having to evolve a single, perfect program.

In situations where the environment gradually changes over time, MGP can be used to adapt the genetic programs without re-evolving the population. Instead only the voting weights of the programs are changed to the updated training data to follow the dynamics of the changing environment. This results in fast adaptation without rediscoverying the programs.

Furthermore, in many real situations the training data is very sparse and/or highly noisy and a single, perfect expert may not exist. The MGP strategy provides a method that exploits the diversity produced by genetic programming. The evolved programs for which so much time and space were invested are used for improving the robustness of final decision making.

The current work can be extended in several directions. We observed that the choice of γ value affects the resulting performance (see Figure 12). In the present work, we did not attempt to optimize the γ values.

In this paper we confined ourselves to the simplest problem formulation in which the output of programs is a binary value. The present method can be extended to the case of real-valued output. From the theoretical point of view, the generalization seems straightforward as is demonstrated in [Vovk, 1994]. In practice, real-valued output is more realistic and can naturally be produced by employing program structures like neural trees [Zhang and Mühlenbein, 1996].

Acknowledgments

This research was supported in part by a grant from the Korea Science and Engineering Foundation (KOSEF) under contract number 961-0901-001-2.

References

[Frankone et al., 1996] F.D. Frankone, P. Nordin, and W. Banzhaf, Benchmarking the generalization capabilities of a compiling genetic programming system using sparse data sets, *Proc. First Annual Conf. on Genetic Programming*, Cambridge, MA: MIT Press, pp. 72-80, 1996.

[Ito et al., 1996] T. Ito, H. Iba, and M. Kimura, Robustness of robot programs generated by genetic programming, *Proc. First Annual Conf. on Genetic Programming*, Cambridge, MA: MIT Press, pp. 321-326, 1996.

[Koza, 1992] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press, 1992.

[Littlestone and Warmuth, 1994]
N. Littlestone and M.K. Warmuth, The weighted majority algorithm, *Information and Computation*, 108:212-261, 1994.

[Reynolds, 1994] C.W. Reynolds, Evolution of obstacle avoidance behavior: using noise to promote robust solutions, *Advances in Genetic Programming*, MIT Press, pp. 221-241, 1994.

[Vovk, 1994] S. Vovk, Aggregating strategies, In *Proc. Third Annual Workshop on Computational Learning Theory*, Morgan Kaufmann, pp. 372-383, 1994.

[Zhang and Mühlenbein, 1996] Byoung-Tak Zhang and Heinz Mühlenbein, Adaptive fitness functions for dynamic growing/pruning of program trees, *Advances in Genetic Programming 2*, P.J. Angeline and K. Kinnear (eds.), MIT Press, Chapter 12, pp. 241-256, 1996.