Genetic Programming and Evolvable Machines, 1, 217–242, 2000

Bayesian Methods for Efficient Genetic Programming

BYOUNG-TAK ZHANG http://scai.snu.ac.kr/ ~ btzhang
Artificial Intelligence Lab (SCAI), School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Korea

Received June 1, 1999; Revised October 6, 1999

Abstract. A Bayesian framework for genetic programming (GP) is presented. This is motivated by the observation that genetic programming iteratively searches populations of fitter programs and thus the information gained in the previous generation can be used in the next generation. The Bayesian GP makes use of Bayes theorem to estimate the posterior distribution of programs from their prior distribution and likelihood for the fitness data observed. Offspring programs are then generated by sampling from the posterior distribution by genetic variation operators. We present two GP algorithms derived from the Bayesian GP framework. One is the genetic programming with the adaptive Occam's razor (AOR) designed to evolve parsimonious programs. The other is the genetic programming with incremental data inheritance (IDI) designed to accelerate evolution by active selection of fitness cases. A multiagent learning task is used to demonstrate the effectiveness of the presented methods. In a series of experiments, AOR reduced solution complexity by 20% and IDI doubled evolution speed, both without loss of solution accuracy.

Keywords: Bayesian genetic programming, probabilistic evolution, adaptive Occam's razor, incremental data inheritance, parsimony pressure, data subset selection

1. Introduction

Genetic programming (GP) is a method for learning the most fit computer programs by means of artificial evolution. The genetic programs are usually represented as trees consisting of functions and terminal symbols [20]. A population of computer programs are generated at random. They are evolved to better programs using genetic operators. The ability of the program to solve the problem is measured as its fitness value. Since Lisp S-expressions can be represented as trees, genetic programming can, in principle, evolve any Lisp programs. Due to this representational power, GP provides a general and powerful tool for automatic programming and machine learning [8]. However, the general applicability of GP suffers from large amounts of *space* and *time* required for generating intermediate solutions.

Space requirements of GP are proportional to the product of population size and the size of each program. Thus, given a fixed population size, space requirements can be reduced by minimizing the program size at each generation. Time requirements of GP are proportional to the product of the population size, individual program size, data size, and the number of generations. Given a fixed population size and generation number, the time complexity of GP can be reduced by minimizing program size and/or data size. As reviewed in Section 2, several methods have so far been proposed and tested to scale up genetic programming with respect to space and time requirements. However, relatively little effort has been made to develop theories to provide principled methods for efficient guidance of evolutionary dynamics of genetic programming.

In this paper, we present principled ways for genetic programming to evolve compact programs as fast as possible without loss of their accuracy. These methods are based on the Bayesian evolutionary framework [41]. In the Bayesian approach to genetic programming, genetic programs are viewed as models of the fitness data. Bayes theorem is used to estimate the posterior probabilities of programs from their prior probabilities and likelihoods for the fitness cases observed. Offspring programs are then generated by sampling from the posterior distribution by using genetic operators.

Two specific methods for Bayesian genetic programming are presented. One is the genetic programming with the adaptive Occam's razor (AOR) designed to evolve parsimonious programs. The other is the genetic programming with incremental data inheritance (IDI) designed to accelerate evolution by active selection of training cases. All these methods are implemented as adaptive fitness functions that take into account the dynamics of evolutionary processes. A multiagent learning task is used to demonstrate the effectiveness of the presented methods.

The paper is organized as follows. In Section 2, we briefly review related works and describe the task that will be used in empirical studies. Section 3 introduces the Bayesian approach to genetic programming and its variants. Sections 4 and 5 provide the Bayesian GP methods for program growth control and acceleration of evolution speed. The performances of the presented methods are also demonstrated. Section 6 discusses the directions for future research.

2. Upscaling genetic programming

Before the Bayesian approach is described, we review previous efforts for scaling up genetic programming. The review is focused on five issues that are most related to the present work (see Table 1): analysis of code growth phenomena, enforcing parsimony pressure, promoting modularity, operator design, and fitness case selection. It should be mentioned that genetic programming is a very broad and rapidly evolving field of research and our review does not attempt to be complete.

2.1. Previous efforts

Genetic programming is distinguished from other evolutionary algorithms in that it uses variable-size representations. Many GP researchers have observed that GP programs tend to rapidly grow in size as the population evolves. This phenomenon is known as "bloat." Angeline [3], for example, observes that many of the evolved solutions contained code segments that, when removed, did not alter the result

Table 1. A summary of previous efforts for upscaling genetic programming

Subjects	Selected references	
	Angeline [3]	Banzhaf [7]
Bloat	Landgon [22]	Nordin [27]
	Wu [39]	Zhang [44]
	Blickle [10]	Iba [18]
Parsimony	Kinnear [19]	Rosca [31]
	Soule [36]	Zhang [44, 45]
	Andre [1]	Angeline [2]
Modularity	Koza [21]	O'Reilly [28]
	Spector [37]	Rosca [32]
	Angeline [4]	Banzhaf [7]
Operators	Chellapilla [11]	Koza [20]
	Luke [24]	Poli [29]
	Gathercole [13, 14]	Hillis [17]
Subset	Schoenauer [33]	Siegel [34]
	Teller [38]	Zhang [40, 42]

This list contains references which are directly related to the present work, and thus it is far from complete.

produced by the solution. Soule et al. [35] observe that removing non-functional codes at every generation does not halt the program's growth. Instead, the programs generate code which, while functional, is never actually executed.

Similar observations have been made in "biological" evolution. That is, introns or non-coding segments emerge in DNA as well. Wu and Lindsay [39] give a recent review of biological introns. Banzhaf et al. [7] characterize introns as having two salient features: An intron is a segment of the genotype that emerges from the process of the evolution of variable length structures, and an intron does not affect the survivability of the individual directly. Introns in GP turned out to be a mixed blessing. On one hand, they may benefit evolution since they enable a genetic program to protect itself against the destructive effect of crossover, and allow the population to preserve highly-fit building blocks [27]. On the other hand, from the practical point of view, introns usually result in run stagnation, poor results, and a heavy drain on memory and CPU time [8, 35].

An explanation for the cause of bloat in GP was provided by Zhang and Mühlenbein [44]. Based on statistical learning theory, they show that the total error of genetic programs can be decomposed into two terms attributed to bias and variance. Since complex models are more expressive and thus better at reducing the bias error than simple models, GP programs tend to grow until they fit the fitness data perfectly unless complex models are penalized to reduce the variance error. Empirical evidence supporting the explanation was given by an analysis of the distributions of fitness vs. complexity of genetic programs generated by a large number of GP runs for solving 7-parity problems.

Recently, Langdon and Poli [22] provide a similar but more general explanation for bloat. They argue that any stochastic search techniques, including GP, will tend to find the most common programs in the search space of the current best fitness.

Since in general there are more of these which are long than there are which are short (but GP starts with the shorter ones) the population tends to be filled with longer and longer programs. Based on this argument, Langdon [23] presents new crossover operators that carefully control variation in size and produce much less bloat.

For the purposes of upscaling genetic programming, there are several reasons for generally preferring parsimonious programs to complex ones. Parsimonious programs typically require less time and less space to run. This is particularly important during the GP process which may need to store and evaluate populations of hundreds or thousands of programs. When hardware implementation of GP solutions is in mind, or when GP evolves hardware circuits online ("evolvable hardware"), simple circuits require less hardware resource and execution time than complex circuits.

In addition, statistical theory says that simpler models are likely to generalize better on unseen fitness cases. As shown in [44], when biases are equal, a simple program has less variance in average than a complex program, resulting in smaller total error. This is the statistical background behind the principle of Occam's Razor [43] and the necessity for parsimony pressure in genetic programming. However, an effective control of program growth is a difficult task, since too much pressure on parsimony (that reduces variance error) may lead to loss of diversity and thus result in low accuracy (greater bias error). The adaptive Occam method [44] was presented as a method for striking a balance between accuracy and parsimony of GP solutions. It adapts the complexity penalty in the fitness function during a run so that programs of minimal complexity are evolved without loss of their accuracy. Several researchers support that parsimony pressure is one of the easiest and effective methods for avoiding bloat [10, 18, 19, 31, 35, 36].

Controlling program growth is one method for reducing time complexity as well as space complexity of genetic programming. Another approach to upscaling GP is by increasing modularity and reusability of programs. Koza [21] introduces the automatically defined functions or ADFs. An ADF is a subroutine that is evolved during a run of genetic programming and which may be called by the main program that is being simultaneously evolved during the same run. He reports that GP with ADFs produced more parsimonious solutions and required fewer fitness evaluation. Angeline and Pollack [2] develop an alternative method called module acquisition (MA). Rosca and Ballard [32] present mechanisms for adaptive representation learning (ARL) that creates new subroutines through discovery and generalization of blocks of code. Spector [37] presents a method for evolving a collection of automatically defined macros (ADMs). He shows that ADMs sometimes provide a greater benefit than ADFs.

The search process of GP can be made more efficient by designing novel genetic operators. Traditionally, crossover has been considered as the primary operator and mutation as the secondary. This view is consistent with the notion that GP is best made by combining building blocks, as hypothesized in bitstring genetic algorithms. Koza [20] has argued that mutations have little utility in GP because of the position-independence of GP subtrees. However, the roles of building blocks and of crossover have become increasingly controversial in recent years. Luke and

Spector [24] experimentally demonstrate that mutation can in fact have utility and that crossover does not consistently have a considerable advantage over mutation. Banzhaf et al. [6] also found that increasing the mutation rate can significantly improve the generalization performance of genetic programming. Angeline [4] shows that mutation operations can perform on par with subtree crossover and suggests that the building block hypothesis may not accurately describe the operational chracteristics of subtree crossover. Chellapilla [11] proposes a method for evolving computer programs without crossover. Poli and Langdon [29] propose various crossover operators and compare their search properties. Haynes [16] demonstrates how small changes in representation, decoding, and evaluation of genetic programs can increase the probability of destructive crossover and mutation while not changing the search space.

More recently, several authors have shown that fitness evaluation in genetic programming can be significantly accelerated by selecting a subset of fitness cases. The basic idea is that, other things being equal, the evolution time can be minimized by reducing the effective data size for each generation. Siegel [34], for example, describes a GP method for evolving decision trees using subsets of given training cases. Each fitness case has a fitness measure and the training cases which tend to be incorrectly classified by decision trees become more fit, and therefore selected more frequently. This method is motivated by competitive selection of fitness cases in the host-parasite model of Hillis [17]. Similar ideas of competitive selection of fitness cases have been refined by Teller and Andre [38] and by Gathercole and Ross [14]. Here, the fitness of a training case is evaluated only when the cost of evaluating another fitness case is outweighed by the expected utility that the new information will provide.

Another class of methods for fitness case selection is based on incremental learning. Schoenauer et al. [33] propose the successive optimization scheme that gradually refines the fitness. Evolution typically starts by considering a unique fitness case, and additional fitness cases are "gradually" taken into account when the current population meets some performance criterion. Though in a slightly different context, Zhang [40] presents a selective incremental learning (SEL) method in which an "incrementally" chosen subset of given training data is used for fitness evaluation of the model. The data sets are divided into two disjoint sets of candidate and training sets. Each candidate data point has a fitness value, called criticality, defined as the error made by the model. Fitter candidates are then incrementally selected into the training set as learning proceeds. SEL evolves a single data set for a single model. The method presented in Section 5 generalizes this to populations of multiple models and multiple data sets.

2.2. *Evolving multiagent cooperation strategies using GP*

A multiagent learning task is used as a testbed for the Bayesian genetic programming methods. In an $n \times n$ grid world, a single table and four robotic agents are placed at random positions, as shown in Figure 1. A specific location is designated as the destination. The goal of the robots is to transport the table to the

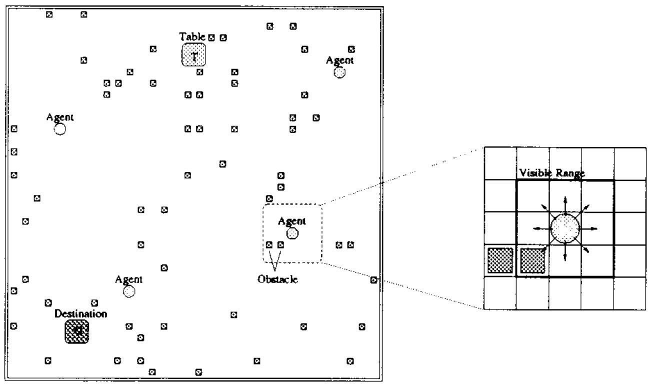

Figure 1. The environment for multiagent learning. On a grid world of 32 × 32, there are four robots, 64 obstacles, and a table. The task of the robots is to transport the table in group motion to the designated target position G.

destination in group motion. The robots need to move in herd since the table is too heavy and large to be transported by single robots. The robots share a common control program A_{best}.

The objective of a GP run is to find a multi-robot algorithm A_{best} that, when executed by the robots in parallel, causes efficient table transport behavior in group. To evolve the programs, the robots activate each candidate program A_i for $i = 1, \dots, M$ in parallel to run a team trial. At the beginning of the trial, the robot locations are chosen at random in the arena. They have different positions and orientations. During a trial, each robot is granted a total of $S_{\max}$ elementary movements. The robot is allowed to stop in less than $S_{\max}$ steps if it reaches the goal. At the end of the trial, each robot i gets a fitness value which was measured by summing the contributions from various factors.

The terminal and function symbols used for building GP trees to solve this problem are listed in Tables 2 and 3. The terminal set consists of six primitive actions: `FORWARD`, `AVOID`, `RANDOM-MOVE`, `TURN-TABLE`, `TURN-GOAL`, and `STOP`. The function set consists of six primitives: `IF-OBSTACLE`, `IF-ROBOT`, `IF-TABLE`, `IF-GOAL`, `PROG2`, and `PROG3`. Each fitness case represents a world of 32 by 32 grid on which there are four robots, 64 obstacles, and the table to be transported (see Figure 1 for an example of fitness cases). A set of 200 training cases that are generated at random is used for evolving the programs. An independent set of 200 random cases is used to test the performance of the evolved programs.

All the robots use the same control program. To evaluate the fitness of robots, we made a complete run of the program for one robot before the fitness of another is measured. The raw fitness value, $e_{i,c}(g)$, of individual i at generation g against case c is computed by considering various factors. These include the distance between the target and the robot, the number of steps moved by the robot, the number of collisions made by the robot, the distance between starting and final position of the robot, and the penalty for moving away from other robots.

Table 2. GP terminal symbols for the multiagent task

Symbol	Description
FORWARD	Move one step forward in the current direction
AVOID	Check clockwise and make one step in the first direction that avoids collision
RANDOM-MOVE	Move one step in the random direction
TURN-TABLE	Make a clockwise turn to the nearest direction of the table
TURN-GOAL	Make a clockwise turn to the nearest direction of the goal
STOP	Stay at the same position

Table 3. GP function symbols for the multiagent task

Symbol	Description
IF-OBSTACLE	Check collision with obstacles
IF-ROBOT	Check collision with other robots
IF-TABLE	Check if the table is nearby
IF-GOAL	Check if the table is nearby
PROG2, PROG3	Evalute two (or three) subtrees in sequence

3. Bayesian genetic programming

In this section we present a theory of genetic programing that is based on Bayesian inference. The general theory is then applied to addressing two important issues in genetic programming, i.e., control of program growth and acceleration of fitness evaluation. This section aims to provide an outline of the Bayesian GP approach

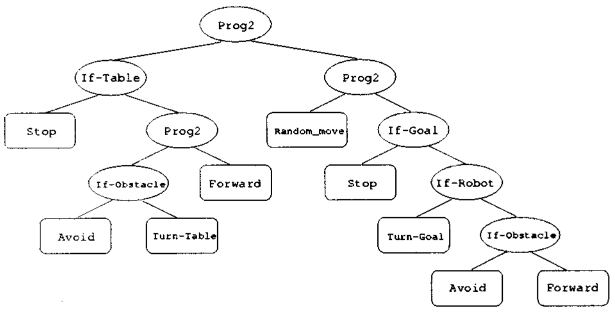

Figure 2. An example genetic program for the multiagent learning task. Non-terminal nodes denote checking sensor inputs of the robots, and terminals indicate actions to be taken. The meaning of the symbols is described in Tables 2 and 3.

and some distinguishing features in different implementations. The algorithms are detailed in the following two sections.

3.1. Bayesian formulation of genetic programming

Genetic programming works by initializing a population of programs and iteratively producing the next generation of fitter programs, i.e. $\mathcal{A}(g) = \{A_i^g\}_{i=1}^M$, where A_i^g denotes the ith program at generation g, and M is the population size. Genetic operators such as mutation and crossover are used to produce offspring programs from the parent programs. New generations are produced repeatedly until the maximum number of generations $g_{\max}$ is reached or some other termination condition is satisfied. The goodness or fitness of a program is measured in terms of a set D of fitness cases or training data and the programs can be considered as a model of the unknown process f generating the data.

In the Bayesian GP, the *best* program is defined as the *most probable* model, given the data D plus the prior knowledge on the problem domain. Bayes theorem provides a direct method for calculating such probabilities [15]. It states that the posterior (i.e. after observing the data D) probability of a program A is

$$P(A \mid D) = \frac{P(D \mid A)P(A)}{P(D)} = \frac{P(D \mid A)P(A)}{\int_{\mathcal{A}} P(D \mid A)P(A)dA}, \tag{1}$$

where $\mathcal{A}$ is the space of *all* possible programs (in case of A taking discrete values, the integral will be replaced by summation). Here $P(A)$ is the prior (i.e. before observing the data) probability distribution for the programs, and $P(D \mid A)$ is the likelihood of the program for the data.

The relationship between the states of models and their probabilities is established by borrowing the concept of “energy” from statistical physics [26]. We regard the GP system as a thermodynamic system. Every possible state s of the system has some definite energy $E(s)$. The system energy can fluctuate and it is assumed that the probability of a (thermodynamic) system being in state s, given that the temperature is T is given as

$$P(s) = \frac{1}{Z}\exp(-E(s)/T), \tag{2}$$

where $E(s)$ is the energy of the system and Z is the normalization constant needed to make the distribution integrate (or sum) to one. This distribution is known as the canonical (or Boltzmann) distribution.

Under the canonical distribution, the prior distribution of programs A can be expressed as

$$P(A) = \frac{1}{Z_A(\alpha)} \exp(-\alpha F_A), \tag{3}$$

where $Z_A(\alpha)$ is a normalizing constant, $\alpha = 1/T$, and F_A is the "energy" of model A in the "equilibrium" state. For example, the energy function, F_A, can be chosen as the total number of nodes in genetic program A. This choice of prior distribution says that we expect the program size to be small rather than large, thus implementing a parsimony pressure. Similarly, the likelihood of models A for the fitness data D can be expressed as

$$P(D \mid A) = \frac{1}{Z_D(\beta)} \exp(-\beta F_D), \tag{4}$$

where F_D is an error function for the fitness set D, β controls the variance of the noise, and $Z_D(\beta)$ is a normalization factor. The likelihood factor gives preference to programs that fit better to (have less error for) the fitness cases.

Initially, the shape of the prior probability distribution of programs $P(A_i)$ is flat to reflect the fact that little is known in advance. Evolution is considered as an iterative process of revising the posterior distribution of models $P(A \mid D)$ by combining the prior $P(A)$ with the likelihood $P(D \mid A)$. In each generation, Bayes theorem (1) is used to estimate the posterior fitness of individuals from their prior fitness values. The posterior distribution $P(A \mid D)$ is then used to generate its offspring.

The objective of Bayesian genetic programming (Figure 3) is to find a program A^g_{best} that maximizes the posterior probability:

$$A^g_{best} = \min_{g \le g_{\max}} \arg \max_{A^g_i \in \mathscr{A}(g)} P_g(A^g_i \mid D) \tag{5}$$

1. Initialize M genetic programs A^1_i, $i = 1, \ldots, M$. Let D of N fitness cases be given. Set generation count $g \leftarrow 1$.
2. Compute posterior probabilities $P_g(A^g_i|D)$, $i = 1, \ldots, M$ using (7).
3. Generate L offspring $A'_k, k = 1, \ldots, L$, by sampling from $P_g(A^g_i|D)$ using genetic operators.
4. Select M parents $A^{g+1}_i, i = 1, \ldots, M$, of generation $g+1$ from $A'_k, k = 1, \ldots, L$.
5. $g \leftarrow g + 1$. If $g \le g_{max}$, go to step 2.

Figure 3. Outline of the Bayesian genetic programming procedure.

where $g_{\max}$ is the maximum number of generations and $\mathscr{A}(g)$ is the populations of size M:

$$\mathscr{A}(g) = \{A_i^g\}_{i=1}^M. \tag{6}$$

The posterior probability $P_g(A_i^g \mid D)$ of program A_i^g is computed with respect to the gth population:

$$P_g(A_i^g \mid D) = \frac{P(D \mid A_i^g)P(A_i^g)}{\sum_{j=1}^M P(D \mid A_j^g)P(A_j^g)}, \tag{7}$$

where $P(D \mid A_i^g)$ is the likelihood and $P(A_i^g)$ is the prior probability of (or degree of belief in) A_i^g. Note that the posterior probability is approximated by a fixed-size population $\mathscr{A}(g)$ which is typically a small subset of the entire program space $\mathscr{A}$.

Genetic operators are applied to generate L offspring A'_k, $k = 1, \dots, L$. Formally, this proceeds in two steps. First, candidates are generated by sampling from the proposal distribution:

$$Q_g(A'_k \mid A_i^g). \tag{8}$$

The specific form of $Q_g(\cdot \mid \cdot)$ is determined by variation operators. For example, for subtree crossover from two parents it is given as:

$$Q_C(A'_k \mid A_i) = \sum_{A_j \in \mathscr{A}(g)} P_S(A_j \mid A_i) P_R(A'_k \mid A_i, A_j). \tag{9}$$

Here $P_S(A_j \mid A_i)$ is the probability of individual A_j in $\mathscr{A}(g)$ being selected as a mate for A_i and $P_C(A'_k \mid A_i, A_j)$ is the probability of A_i and A_j producing A'_i by crossover.

Then, each candidate generated by genetic operators is accepted with probability

$$a_g(A'_k \mid A_i^g) = \min\left\{1, \frac{P_g(A'_k \mid D)}{P_g(A_i^g \mid D)}\right\}, \tag{10}$$

where $P_g(A_i^g \mid D)$ is computed by (7) and $P_g(A'_k \mid D)$ is the posterior probability of A'_k estimated with respect to the current population:

$$P_g(A'_k \mid D) = \frac{P(D \mid A'_k)P(A'_k)}{\sum_{j=1}^M P(D \mid A_j^g)P(A_j^g)}. \tag{11}$$

If A'_k is rejected in (10), then A_i^g is retained, i.e., $A'_k \leftarrow A_i^g$. Note that this acceptance function does not exclude the case that A'_k is generated by crossover from A_i^g and another parent $A_j^g \in \mathscr{A}(g)$, $j \neq i$.

After L offspring A'_k, $k = 1, \dots, L$, are generated, M of them are selected to build the new population:

$$\mathscr{A}(g+1) = \{A_i^{g+1}\}_{i=1}^{M}. \tag{12}$$

This defines the posterior distribution $P_{g+1}(A_i^g \mid D)$ at the next generation. It should be mentioned that this formulation of offspring selection is intentionally very general so that it can accommodate various forms of existing selection schemes, such as (μ, λ) selection [5, 25].

In effect, the evolutionary inference step from generation g to $g+1$ is considered to induce a new fitness distribution $P_{g+1}(A_i^g \mid D)$ from priors $P(A_i^g)$ through posterior distribution $P_g(A_i^g \mid D)$ following Bayes formula, using genetic operators. Based on this theoretical framework we present in the following subsections two examples of Bayesian genetic programming that employ specific techniques for effective control of evolutionary dynamics. Detailed procedures and experimental results are described in Sections 4 and 5.

3.2. *GP with the adaptive Occam's razor*

The first Bayesian GP focuses on the fact that genetic programming iteratively searches populations of more probable (more likely in terms of the data and the priors) programs and thus the information gained in the previous generation can be used in the next generation to revise the prior (before seeing the data) belief in true programs. Thus, the posterior distribution can be written in the form

$$P_g(A_i^g \mid D) = \frac{P(D \mid A_i^g) P_{g-1}(A_i^g)}{\sum_{j=1}^{M} P(D \mid A_j^g) P_{g-1}(A_j^g)}, \tag{13}$$

where the priors $P_{g-1}(A_j^g)$ are now expressed explicitly as a function of generation rather than the fixed prior $P(A_i^g)$ as in (7). After computing the posterior probabilities for A_i^g, the priors $P_{g-1}(A)$ are revised to $P_g(A)$ by a belief update function $u(\cdot, \cdot)$:

$$P_g(A) = u\big(P_{g-1}(A), P_g(A_i^g \mid D)\big). \tag{14}$$

An implementation of $u(\cdot, \cdot)$ will be described in the next section.

Basically, the Bayesian GP starts with generating programs according to an initial prior distribution $P_0(A)$ on the program sizes. Typically, the prior distribution is given as uniform, reflecting the fact that little is known in advance about the optimal size of genetic programs. Then, the fitness $F_i(g)$ of the programs is measured on the training cases, which results in the estimation of the likelihood $P(D \mid A_i^g)$ of programs (more details in Section 4). Combining the prior and the likelihood by Bayes formula, we get the posterior probabilities $P_g(A_i^g \mid D)$. The current prior is then updated to reflect the new information gained from the posterior distribution.

Note that we assign prior distributions on the complexity of models (programs). In addition, information theory [12] says that the probability and code length (complexity) of models are related as $L(A) = -\log P(A)$. By making use of this, the complexity of programs can be controlled to evolve parsimonious and accurate programs. In fact, we show in Section 4 that the adaptive Occam method presented in [44] is derived from the Bayesian genetic programming framework.

3.3. *GP with incremental data inheritance*

In the second example of the Bayesian approach to GP, we make use of the fact that the Bayes formula suggests an incremental, evolutionary learning rule: infer programs of higher posterior probability from the existing programs by observing new fitness cases. This leads to writing the posterior distribution as

$$P_g(A_i^g \mid D_i^g) = \frac{P(D_i^g \mid A_i^g) P_{g-1}(A_i^g \mid D_i^{g-1})}{\sum_{j=1}^{M} P(D_j^g \mid A_j^g) P_{g-1}(A_j^g \mid D_j^{g-1})}, \tag{15}$$

where the data set D_i^g is now a variable of generation number g and specific to a single program A_i^g. The collection of D_i^g constitutes the data population $\mathscr{D}(g)$. The observation of new data will lead to update of the prior distribution:

$$P_g(A \mid D_i^g) = u\left(P_{g-1}(A \mid D_i^{g-1}), P_g(A_i^g \mid D_i^g)\right). \tag{16}$$

The revision of prior distribution is the same as the process for equation (14), except that the data D_i^{g-1} for the estimation of likelihood (and thus the posterior probabilities of programs) is now a function of generation rather than fixed as D.

The genetic programming with the incremental data inheritance (IDI) method [42] is an example of this approach, where D_i^g specifically satisfies the following conditions:

$$D_i^g \subset D, \qquad |D_i^{g-1}| < |D_i^g|, \tag{17}$$

where D is the entire data set given for training. In this method, the fitness of programs is estimated on incrementally chosen data subsets D_i^g, rather than on the whole data set D, and thus the evolution is accelerated by reducing the effective number of fitness evaluations. More details are described in Section 5.

4. Bayesian GP for parsimonious solutions

Statistical theories suggest that models which are too simple lack sufficient learning capability while models which are too complex may generalize poorly on unseen data. As reviewed in Section 2.1, several researchers have observed that the program size tends to grow without bound or "bloat" (see, for example, [22]). In this section we describe a Bayesian method for evolving parsimonious programs.

4.1. *Algorithm description*

The GP algorithm for the adaptive Occam method (Figure 4) is the same as that of the Bayesian GP procedure described in Figure 3, except three differences. The first is the raw fitness calculated in step 2. The derivation of the fitness function is given in Section 4.2. The second difference is in step 3, where the posterior probability (13) is substituted for (7); in (13) the prior is revised each generation while in (7) it is constant. The third is the additional step 6 for the revision of priors.

4.2. *Fitness evaluation*

For a convenient implementation of the Bayesian evolutionary algorithm we take the negative logarithm of the posterior probability $P_g(A_i^g \mid D)$ and use it as the fitness function

$$F_i(g) = -\log P_g(A_i^g \mid D), \tag{18}$$

```
1. Initialize $M$ genetic programs $A_i^1$ for $i = 1,...,M$. Let $D$ of $N$ fitness cases be given. Set $g \leftarrow 1$.
2. Compute raw fitness $F_i(g) = E_i(g) + \alpha(g)C_i(g)$, $i = 1,...,M$.
3. Compute posterior probabilities $P_g(A_i^g|D)$, $i = 1,...,M$ using (13).
4. Generate $L$ offspring $A'_k, k = 1,...,L$, by sampling from $P_g(A_i^g|D)$ using genetic operators.
5. Select $M$ parents $A_i^{g+1}, i = 1,...,M$, of generation $g+1$ from $A'_k, k = 1,...,L$.
6. Revise prior distribution $P_{g-1}(A)$ to $P_g(A)$.
7. $g \leftarrow g + 1$. If $g \leq g_{max}$, go to step 2.
```

Figure 4. Outline of the Bayesian genetic programming with the adaptive Occam's razor (AOR).

where $A_i^g \in \mathscr{A}(g)$. Then the evolutionary process is reformulated as a minimization process

$$A_{best}^g = \min_{g \le g_{\max}} \arg \min_{A_i^g \in \mathscr{A}(g)} F_i(g), \tag{19}$$

where the fitness function is expressed as

$$F_i(g) = -\log P(D \mid A_i^g) - \log P(A_i^g). \tag{20}$$

As described in the previous section, we can write the likelihood function in Bayes' theorem (1) in the form [9]

$$P(D \mid A) = \frac{1}{Z_D(\beta)} \exp(-\beta F_D) \tag{21}$$

where F_D is an error function, β controls the variance of the noise, and $Z_D(\beta)$ is a normalization factor. If we assume that the data has additive zero-mean Gaussian noise, then the probability of observing a data value y for a given input vector $\mathbf{x}$ would be

$$P(y \mid \mathbf{x}, A_i^g) = \frac{1}{Z_D(\beta)} \exp\left(-\frac{\beta}{2}\left(f(\mathbf{x}; A_i^g) - y\right)^2\right) \tag{22}$$

where $Z_D(\beta)$ is a normalizing constant. Provided the data points are drawn independently from this distribution, we have

$$P(D \mid A_i^g) = \prod_{c=1}^{N} P(y_c \mid \mathbf{x}_c, A_i^g) \tag{23}$$

$$= \frac{1}{Z_D(\beta)} \exp(-\beta F_D). \tag{24}$$

where $(\mathbf{x}_c, y_c) \in D$ are training cases and F_D is given as

$$F_D = E(D \mid A_i^g) = \frac{1}{2} \sum_{c=1}^{N} \left(f(\mathbf{x}_c; A_i^g) - y_c\right)^2. \tag{25}$$

If we also assume that a Gaussian prior on the architecture of program A_i^g, we have

$$P(A_i^g) = \frac{1}{Z_A(\alpha)} \exp(-\alpha F_A) \tag{26}$$

where $Z_A(\alpha)$ is a normalizing constant. For example, F_A can be chosen in the form

$$F_A = C(A_i^g) = \frac{1}{2}\sum_{k=1}^{K}\theta_k^2 \tag{27}$$

where θ_k are the parameters defining the program A_i^g. This choice of prior distribution says that we expect the complexity parameters to be small rather than large, thus implementing a parsimony pressure.

Substituting (24) and (26) into (20), the fitness function can be expressed as

$$F_i(g) = \beta F_D + \alpha F_A \tag{28}$$

$$= \beta E(D \mid A_i^g) + \alpha C(A_i^g), \tag{29}$$

where the first term reflects the error and the second the model complexity.

The exact calculation of the constants β and α in equation (29) 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

$$F_i(g) = E_i(g) + \alpha(g)C_i(g), \tag{30}$$

where $E_i(g)$ and $C_i(g)$ are the measures for error and complexity of the program, and the parameter β is absorbed into the adaptive parameter $\alpha(g)$ which balances the error and complexity factors as follows:

$$\alpha(g) = \begin{cases} \dfrac{1}{N^2}\dfrac{E_{best}(g-1)}{\hat{C}_{best}(g)} & \text{if } E_{best}(g-1) > \epsilon \\ \dfrac{1}{N^2}\dfrac{1}{E_{best}(g-1)\cdot\hat{C}_{best}(g)} & \text{otherwise.} \end{cases} \tag{31}$$

This is the adaptive Occam method [44]. User-defined constant ϵ specifies the maximum training error allowed for the run. $E_{best}(g-1)$ is the error of the best program of generation $g-1$. $\hat{C}_{best}(g)$ is the size of the best program at generation g estimated at generation $g-1$. These are used to balance the error and complexity terms to obtain programs as parsimonious as possible while not sacrificing their accuracy. The procedure for estimating $\hat{C}_{best}(g)$ is given in [44].

4.3. *Discussion*

The necessity and difficulty of non-coding segments or introns in genetic programming have been studied by many authors [22, 27, 39]. The adaptive Occam method deals with the non-coding segment problem using a two-phase strategy by means of an adaptive fitness function. In the first stage (during $E_{best}(g-1) > \epsilon$), the growth of non-coding segments is encouraged to increase the diversity of partial solutions. In the second stage (during $E_{best}(g-1) \leq \epsilon$), a strong parsimony pressure is enforced to prefer compact solutions. The transfer from the first stage to the second is controlled by Bayesian inference under the constraint of the user-defined parameter ϵ.

Note that the posterior probability $P_g(A_i^g \mid D)$ can be computed from the fitness values $F_i(g)$ by taking its exponential function:

$$P_g(A_i^g \mid D) = \frac{P(D \mid A_i^g) P_{g-1}(A_i^g)}{\sum_{j=1}^{M} P(D \mid A_j^g) P_{g-1}(A_j^g)} = \frac{\exp(-E_i(g) - \alpha(g) C_i(g))}{\sum_{j=1}^{M} \exp(-E_j(g) - \alpha(g) C_j(g))} \tag{32}$$

This shows that the revision of priors $P_{g-1}(A_i^g)$ is implemented as the update of $\alpha(g)$ in the adaptive Occam method since the priors are reflected in $\hat{C}_{best}(g)$ which leads to revision of $\alpha(g)$.

We also note that minimization of $F_i(g)$ is equivalent to the minimum description length (MDL) principle [18, 30]: the best model is a model whose total code length for model description $L(A_i^g \mid D_i^g)$ and data description $L(D_i^g \mid A_i^g)$ are minimal. In information theory [12] the optimal description length for a model is given as the negative logarithm of probability of the model:

$$L(A_i^g) = -\log P(A_i^g). \tag{33}$$

Similarly, the code length for the data given the program A_i^g is given as:

$$L(D \mid A_i^g) = -\log P(D \mid A_i^g)\}. \tag{34}$$

$L(D \mid A_i^g)$ and $L(A_i^g)$ correspond to the two terms in (20).

4.4. *Experimental results*

The adaptive Occam method for complexity control was applied to the multiagent learning task. Experiments have been performed using the parameter values listed in Table 4. The terminal set and function set consist of six primitives, respectively, as given in Tables 2 and 3. A set of 200 training cases was used for evolving the programs. An independent set of 200 cases was used for evaluating the generalization performance of evolved programs.

The $E_i(g)$-values of program i at generation g are measured as the average of its raw fitness values $e_{i,c}(g)$ (less is better) for the cases c:

$$E_i(g) = \frac{1}{N} \sum_{c=1}^{N} e_{i,c}(g), \tag{35}$$

Table 4. Parameters used in the experiments for GP with the adaptive Occam's razor

Parameter	Value
Population size	100
Max generation	30
Crossover rate	0.9
Mutation rate	0.1
Training cases	200
Test cases	200

where N is the number of fitness cases in the training set. Each $e_{i,c}(g)$-value was computed by considering such factors as the distance between the target and the robot, the number of collisions made by the robot, and the distance between starting and final position of the robot. Note that in the multiagent task there is no target output given for the training case. The goodness of a program is measured by scores (or penalties) it collects by running the robots.

The complexity of a GP tree is defined as the size and depth of the tree:

$$C_i(g) = \text{size}(A_i^g) + \kappa \cdot \text{depth}(A_i^g), \tag{36}$$

where κ is a constant. We used $\kappa = 2$.

Figure 5 compares the fitness values for GP with and without AOR, averaged over 10 runs. Shown are the $E_i(g)$-values of the best individuals in each generation for each method. A tendency can be observed that the GP with the adaptive Occam's razor converges slightly faster than the baseline GP, though there is no significant difference in their final fitness. Figure 6 compares the typical changes of program complexity for both methods. The tree complexity was measured in terms of the tree size plus 2 × depth.

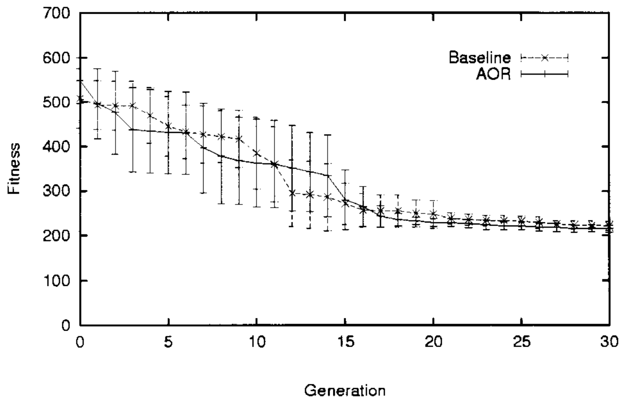

Figure 5. Comparison of fitness values ($E_i(g)$-component only) of the genetic programs for the multiagent task. A tendency can be observed that the GP with the adaptive Occam's razor converges slightly faster than the baseline GP, though there is no significant difference in their final fitness.

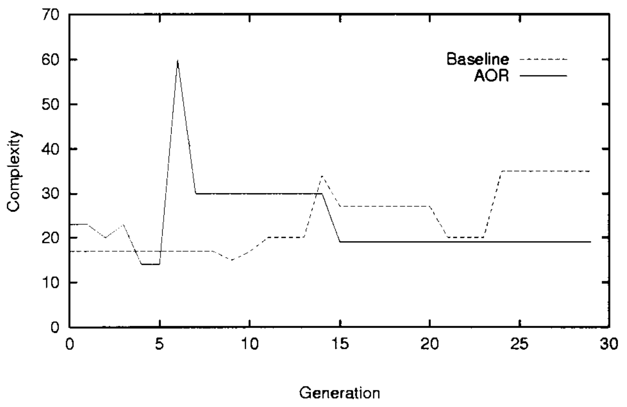

Figure 6. Evolution of the complexity of GP trees for the multiagent task. The GP with the adaptive Occam's razor (AOR) promotes the trees to grow when significant fitness (error) reduction is required, while it prefers smaller trees to larger ones when their fitness (error) is comparable. In contrast, the GP without the Occam factor tends to grow as generation goes on.

More detailed results are summarized in Table 5. The GP with AOR achieved, on average, a 20% reduction of program complexity without loss of solution accuracy (in fact, with a slight improvement both in training and test performances). It can be concluded that the adaptive Occam method evolves smaller programs without loss of generalization capability of the programs.

5. Bayesian GP for accelerated evolution

5.1. Algorithm description

The algorithm for incremental data inheritance (Figure 7) is the same as the GP with AOR, except that the training set D_i^g now increases with generation. The additional step for this modification is step 5, where the training set grows by inheritance. The next section details the data inheritance procedure.

Table 5. Effects of the adaptive Occam's razor (AOR) on the complexity and average fitness values of GP programs for the multiagent learning task

		Average Fitness	
Method	Complexity	Training	Test
Baseline	31.4 ± 6.5	222.1 ± 10.3	228.7 ± 12.3
AOR	25.0 ± 5.5	215.8 ± 8.3	221.3 ± 7.6

The sizes of training and test sets were 200, respectively. The values are averaged over ten runs.

```
1. Initialize $M$ genetic programs $A_i^1$, $i = 1, ..., M$.
   Initialize $D_i^1$ of $N_1$ fitness cases for $i = 1, ..., M$.
   Set $g \leftarrow 1$.
2. Compute fitness values $F_i(g) = E_i(g) + \alpha(g)C_i(g)$, $i = 1, ..., M$.
3. Compute posterior probabilities $P_g(A_i^g|D_i^g)$, $i = 1, ..., M$ using (15).
4. Generate $L$ offspring $A_k', k = 1, ..., L$, by sampling from $P_g(A_i^g|D_i^g)$ using genetic operators.
5. For $i = 1, ..., M$, inherit $D_i^{g+1}$ from the parents $D_i^g$ and $D_j^g$.
6. Revise prior distribution $P_{g-1}(A|D_i^{g-1})$ to $P_g(A|D_i^g)$.
7. $g \leftarrow g + 1$. If $g \leq g_{max}$, go to step 2.
```

Figure 7. Outline of the Bayesian genetic programming with incremental data inheritance (IDI).

5.2. Data inheritance procedure

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. We describe a variant of uniform crossover that we call uniform data crossover. A simplified example for illustrating this process is given in Figure 8.

First, 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+1} and D_j^{g+1}. Second, the data of parents' are mixed into a union set

$$D_{i+j}^g = D_i^g \cup D_j^g, \tag{37}$$

which is then redistributed to two offspring D_i^{g+1} and D_j^{g+1}, where the size of offspring data sets is 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 maintain the diversity of the training data during inheritance we import some portion of data from the base set. The import rate r_i is given as

$$r_i = \rho \cdot (1 - d_i), \qquad 0 \leq \rho \leq 1. \tag{38}$$

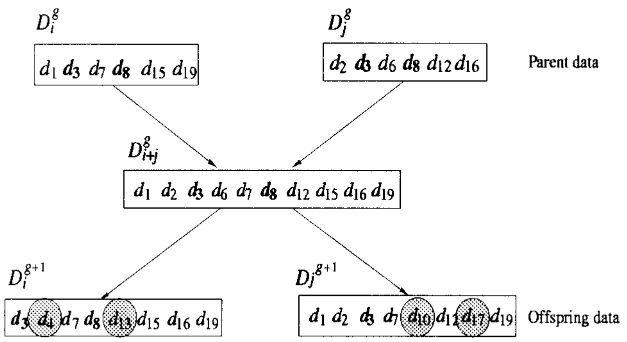
D_i^g
$d_1\ d_3\ d_7\ d_8\ d_{15}\ d_{19}$
D_j^g
$d_2\ d_3\ d_6\ d_8\ d_{12}\ d_{16}$
Parent data
D_{i+j}^g
$d_1\ d_2\ d_3\ d_6\ d_7\ d_8\ d_{12}\ d_{15}\ d_{16}\ d_{19}$
D_i^{g+1}
$d_3\ d_4\ d_7\ d_8\ d_{13}\ d_{15}\ d_{16}\ d_{19}$
D_j^{g+1}
$d_1\ d_2\ d_3\ d_7\ d_{10}\ d_{12}\ d_{17}\ d_{19}$
Offspring data

Figure 8. An illustrative example for data inheritance in IDI. Two parent data sets, D_i^g and D_i^g, are merged to form the union set, D_{i+j}^g, which is then inherited to two offspring data sets, D_i^{g+1} and D_j^{g+1}. The off-spring data points with shaded circles are the examples imported from the base data set D to maintain the diversity of data sets.

where ρ is a constant for import strength. The diversity d_i is measured as the ratio of distinctive examples in the union set:

$$d_i = \frac{|D_{i+j}^g|}{|D_i^g|} - 1, \qquad 0 \le d_i \le 1. \tag{39}$$

Figure 8 illustrates the process of data inheritance, 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.

5.3. *Fitness evaluation*

As in the implementation of the Bayesian GP with the adaptive Occam's razor, we take the negative logarithm of $P_g(A_i^g \mid D)$ and use it as the fitness function

$$F_i(g) = -\log P_g(A_i^g \mid D_i^g), \tag{40}$$

where $A_i^g \in \mathscr{A}(g)$, $D_i^g \subset D$, and $P_g(A_i^g \mid D_i^g)$ are defined by (15):

$$P_g(A_i^g \mid D_i^g) = \frac{P(D_i^g \mid A_i^g) P_{g-1}(A_i^g \mid D_i^{g-1})}{\sum_{j=1}^M P(D_j^g \mid A_j^g) P_{g-1}(A_j^g \mid D_j^{g-1})}. \tag{41}$$

Then the evolutionary process is reformulated as a minimization process

$$A_{best}^g = \min_{g \le g_{\text{mam}}} \arg \min_{A_i^g, D_i^g} F_i(g), \tag{42}$$

where the fitness function is expressed as

$$F_i(g) = -\log P(D_i^g \mid A_i^g) - \log P_{g-1}(A_i^g \mid D_i^{g-1}). \tag{43}$$

Using the similar arguments as in Section 4.2 we can write the likelihood function and the prior distribution of programs in the form

$$P(D_i^g \mid A_i^g) = \frac{1}{Z_D(\beta)} \exp(-\beta F_D) \tag{44}$$

$$P_{g-1}(A_i^g \mid D_i^{g-1}) = \frac{1}{Z_A(\alpha)} \exp(-\alpha F_A) \tag{45}$$

where $Z_D(\beta)$ *and* $\mathrm{Z_A}(\alpha)$ *are normalizing constants.*

We define an adaptive fitness function in its most general form as

$$F_i(g) = E_i(g) + \alpha(g) C_i(g), \tag{46}$$

where $E_i(g)$ and $C_i(g)$ represents the error factor F_D and complexity factor F_A with α and β absorbed into $\alpha(g)$. The parameter $\alpha(g)$ balances these two factors as follows

$$\alpha(g) = \begin{cases} \dfrac{1}{N_g^2} \dfrac{E_{best}(g-1)}{\hat{C}_{best}(g)} & \text{if } E_{best}(g-1) > \epsilon \\ \dfrac{1}{N_g^2} \dfrac{1}{E_{best}(g-1) \cdot \hat{C}_{best}(g)} & \text{otherwise.} \end{cases} \tag{47}$$

This is a generalization of the adaptive Occam method in that N_g is now a variable, rather than fixed value N, and scheduled to increase monotonically as a function of generation g.

5.4. *Discussion*

The incremental data inheritance method has interesting properties that characterize the stability of the algorithm as generation goes on. It maintains a growing subset of given fitness cases for each program. The monotonic growth of the fitness subset ensures that the IDI method eventually achieves the same level of performance as that of the baseline algorithm. This is contrasted with many of existing methods for fitness case selection, including LEF [14] and RAT [38]. Maintaining a separate subset for each program allows the learned component of the program can be retained during evolution. At the same time, the import of new fitness cases from the base set allows the programs to learn new situations, thus leading to gradual improvement in performance.

5.5. *Experimental results*

We compare the performance of the GP with incremental data inheritance (IDI) to the baseline GP, i.e. one that uses the complete training cases from the outset. Experiments have been performed using the parameter values listed in Table 6. GP

Table 6. Parameters used in the experiments for GP with incremental data inheritance

Parameter	Value
Population size	100
Max generation	30
Crossover rate	0.9
Mutation rate	0.1
Training cases	200
Test cases	200
Initial data size (N_0)	20
Data increment size (λ)	6

runs with IDI used $20 + 6g$ examples for each generation g selected from the given data set, i.e. $N_0 = 20$, $\lambda = 6$, for fitness evaluation. For all methods, a total of 200 training cases were used for training and an independent set of 200 test cases was used for evaluating the generalization performance of evolved programs. Fitness factors $E_i(g)$ and $C_i(g)$ are the same as in the experiments for GP with the adaptive Occam's razor, except that the training set size for $E_i(g)$ is now N_g instead of N.

Results are shown in two different forms. Figure 9 shows the evolution of fitness as a function of the generation number, in which there is no significant difference in the performance. Since the GP with IDI uses variable data size, computing time at each generation should be measured by a product of the population size and the data size. This result is shown in Figure 10, where IDI achieved a speed-up factor of approximately four compared with the baseline GP. More detailed results are summarized in Table 7. The incremental data inheritance (IDI) method used only 50% of the time required for the baseline GP. It is interesting to see that, despite

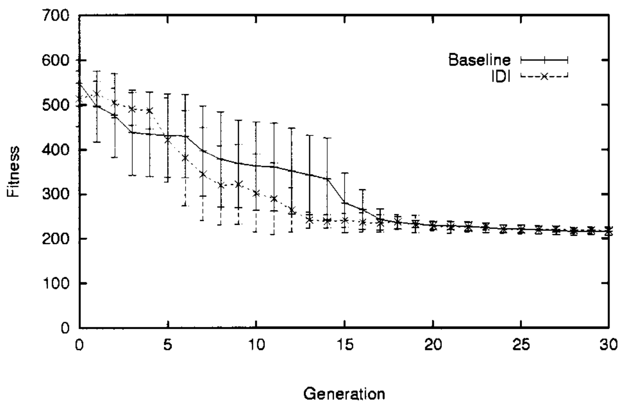

Figure 9. Comparison of fitness values as a function of the generation number. The curves are mean values over ten runs. The GP with IDI shows no loss of fitness values compared to that of the baseline GP algorithm.

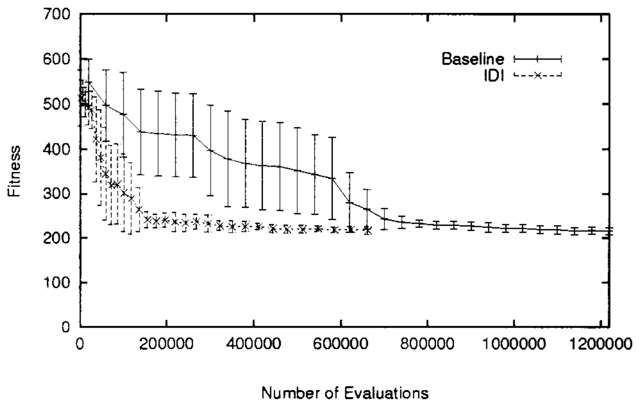

Figure 10. Comparison of fitness values as a function of the number of function evaluations. The curves are mean values over ten runs. The GP with IDI converges much faster than the baseline GP algorithm.

Table 7. Effects of incremental data inheritance (IDI) on the time and average fitness values (lower is better) of GP programs for the multiagent learning task. The sizes of training and test sets were 200, respectively. The values are averaged over ten runs. Time is measured as the total number of fitness evaluations. Also shown are the standard deviations.

		Average Fitness	
Method	Time	Training	Test
Baseline	1,220,000	215.8 ± 8.3	221.3 ± 7.6
IDI	662,000	217.4 ± 9.4	219.9 ± 13.5

the reduced data size, the generalization performance of the GP with IDI was slightly better than that of the baseline GP.

6. Concluding remarks

We have presented a Bayesian framework for genetic programming. Two specific GP algorithms were derived from this framework for reducing the time and space complexity of genetic programming. Applied to the multiagent learning task, the first method, i.e. GP with the adaptive Occam's razor (AOR), achieved approximately 20% reduction of program complexity without any loss in fitness values. The second Bayesian approach to genetic programming, i.e. the incremental data inheritance (IDI) method, used only 50% of the time required for the baseline GP to achieve the same or a little better fitness level.

Though this improvement is significant, it should be mentioned that there is still much room for further reduction of space/time complexity of genetic programming. Future work should address the following issues, among others. One is incorporating better genetic operators. We have focused in the present work on dynamics at the phenotypic level. Further improvement can be achieved by finding more "intelligent" variation operators that adapt to the dynamics at the genotypic level. In terms of Bayesian genetic programming, this involves adapting proposal functions. The second issue is to improve modularity of genetic programs by automatically designing and adapting reusable submodules such as ADFs or libraries. This, combined with the Occam's razor, will further improve the comprehensibility and reusability of genetic programs as well as speed up the GP process.

From the theoretical point of view, the Bayesian framework for genetic programming provides a number of important features. One is that, by formulating the GP process as Bayesian inference, principled techniques for driving evolutionary dynamics of conventional GPs can be developed. In addition to the methods presented in this paper, one can think of other methods within the framework. For example, decision-making can be made more robust by combining multiple programs instead of a single GP tree. The Bayesian GP framework provides a principled way to combine multiple programs to build a committee machine.

Another important feature of Bayesian inference is that it allows background knowledge in the problem domain to be incorporated in a formal way. For instance, if we can guess the distribution of specific function symbols for good GP trees, this knowledge can be reflected in the application probabilities of genetic operators. Background knowledge is important especially for solving real-life problems of practical interest.

Finally, the Bayesian analysis of genetic programming appears to be a useful tool for incorporating various genetic programming procedures into a uniform framework. A theoretical framework is crucial for the design and comparative analysis of various genetic programming algorithms.

Acknowledgments

This research was supported by the Korea Science and Engineering Foundation (KOSEF) Grant# 981-0920-350-2, by the Korea Research Foundation (KRF) Grant# 1998-001-E01025, and by the Korea Ministry of Science and Technology through KISTEP Grant# BR-2-1-G-06. Thanks to Wolfgang Banzhaf and three anonymous reviewers for helpful comments that improved the readability of the paper.

References

1. D. Andre, "Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them," In K. E. Kinnear, Jr. (ed), Advances in Genetic Programming, MIT Press: Cambridge, MA, 1994, Chapter 23, pp. 477–494.
2. P. J. Angeline and J. B. Pollack, "Coevolving high-level representations, in Artificial Life III, C. G. Langton (ed.), Addison-Wesley: Reading, MA, 1993.
3. P. J. Angeline, "Genetic programming and emergent intelligence," in K. E. Kinnear, Jr. (ed.), Advances in Genetic Programming, MIT Press: Cambridge, MA, 1994. Chapter 4, pp. 75–98.

4. P. J. Angeline, “Subtree crossover: Building block engine or macromutation?” in J. R. Koza, et al. (eds.), The Second Genetic Programming Conf. (GP-97), Morgan Kaufmann, San Francisco, CA, 1997, pp. 9–17.
5. T. Bäck, Evolutionary Algorithms in Theory and Practice, Oxford, UK: Oxford University Press, 1996.
6. W. Banzhaf, F. Francone, and P. Nordin, “The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets,” In Proc. 4th Int. Conf. on Parallel Problem Solving from Nature (PPSN-96), W. Ebeling, I. Rechenberg, H.-P. Schwefel, H. M. Voigt (eds.), Springer: Berlin, 1996, pp. 300–309.
7. W. Banzhaf, P. Nordin, and F. Francone, “On some emergent properties of variable size evolutionary algorithms,” in ICGA-97 Workshop on Evolutionary Computation with Variable-Size Representation, 1997, http://www.ai.mit.edu/people/unamay/icga-ws.html.
8. W. Banzhaf, P. Nordin, R. Keller, and F. Francone, Genetic Programming: An Introduction, Morgan Kaufmann: San Francisco, CA.
9. C. M. Bishop, Neural Networks for Pattern Recognition. Oxford University Press: Oxford, UK, 1995.
10. T. Blickle, “Evolving compact solutions in genetic programming: A case study,” in H.-M. Voigt et al. (eds.), Parallel Problem Solving from Nature IV, Springer-Verlag: Berlin, 1996, pp. 564–573.
11. K. Chellapilla, “Evolutionary programming with tree mutations: Evolving computer programs without crossover,” in J. R. Koza, et al. (eds.), The Second Genetic Programming Conf. (GP-97), Morgan Kaufmann: San Francisco, CA, 1997, pp. 431–438.
12. T. M. Cover and J. A. Thomas, Elements of Information Theory, Wiley: New York, 1991.
13. C. Gathercole and P. Ross, “Dynamic training subset selection for supervised learning in genetic programming,” in Parallel Problem Solving from Nature III, Y. Davidor, et al. (Eds.), Springer-Verlag: Berlin, 1994, pp. 312–321.
14. C. Gathercole and P. Ross, “Small populations over many generations can beat large populations over few generations in genetic programming,” in J. R. Koza, et al. (eds.), The Second Genetic Programming Conf. (GP-97), Morgan Kaufmann: San Francisco, CA, 1997, pp. 111–118.
15. A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin, Bayesian Data Analysis, Chapman & Hall: London, 1995.
16. T. Haynes, “Perturbing the representation, decoding, and evaluation of chromosomes,” in J. R. Koza, et al. (eds.), The Third Genetic Programming Conf. (GP-98), Morgan Kaufmann: San Francisco, CA, 1998, pp. 122–127.
17. D. Hillis, “Co-evolving parasites improves simulated evolution as an optimization procedure,” in Artificial Life II, C. Langton, et al. (Eds.), Addison-Wesley: Reading, MA, 1992, pp. 313–324.
18. H. Iba, H. de Garis, and T. Sato, “Genetic programming using a minimum description length principle,” in K. E. Kinnear, Jr. (ed.), Advances in Genetic Programming, MIT Press: Cambridge, MA, 1994, Chapter 12, pp. 265–284.
19. K. E. Kinnear, Jr. “Generality and difficulty in genetic programming: Evolving a sort,” in Proc. of 5th Int. Conf. on Genetic Algorithms (ICGA-93), S. Forrest (ed.), Morgan Kaufmann: San Francisco, CA, 1993, pp. 287–294.
20. J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press: Cambridge, MA, 1992.
21. J. R. Koza, “Scalable learning in genetic programming using automatic function definition,” in K. E. Kinnear, Jr. (ed.), Advances in Genetic Programming, MIT Press: Cambridge, MA, 1994, Chapter 5, pp. 99–117.
22. W. B. Langdon and R. Poli, “Fitness causes bloat: Mutation,” in W. Banzhaf, R. Poli, M. Schoenauer, and T. Fogarty (eds.), The First European Workshop on Genetic Programming (EuroGP’98), Paris, LNCS 1391, Springer-Verlag: Berlin, 1998, pp. 37–48.
23. W. B. Langdon, “Size fair and homologous tree crossovers,” Genetic Programming and Evolvable Machines, vol. 1(1), pp. 95–119, 2000.
24. S. Luke and L. Spector, “A comparison of crossover and mutation in genetic programming,” in J. R. Koza, et al. (eds.), The Second Genetic Programming Conf. (GP-97), Morgan Kaufmann: San Francisco, CA, 1997, pp. 240–248.

25. H. Mühlenbein and D. Schlierkamp-Voosen, "The science of breeding and its application to the breeder genetic algorithm," Evolutionary Computation, vol. 1(4) pp. 335–360, 1994.
26. R. M. Neal, "Probabilistic inference using Markov chain Monte Carlo methods," Technical Report CRG-TR-93-1, Dept. of Computer Science, University of Toronto, 1993.
27. P. Nordin, F. Francone, and W. Banzhaf, "Explicitly defined introns and destructive crossover in genetic programming," in P. J. Angeline and K. E. Kinnear, Jr. (eds.), Advances in Genetic Programming 2, MIT Press: Cambridge, MA, 1996, pp. 111–134.
28. U.-M. O'Reilly, "Investigating the generality of automatically defined functions," in The First Genetic Programming Conf. (GP-96), J. R. Koza (eds.), Morgan Kaufmann: San Francisco, CA, 1996, pp. 351–356.
29. R. Poli, and W. B. Langdon, "On the search properties of different crossover operators in genetic programming," The Third Genetic Programming Conf. (GP-98), Morgan Kaufmann: San Francisco, CA, 1998, pp. 293–301.
30. J. Rissanen, "Stochastic complexity and modeling," Ann. Statist. vol. 14, pp. 1080–1100, 1986.
31. J. P. Rosca, "Analysis of complexity drift in genetic programming," in The Second Genetic Programming Conf. (GP-97), J. R. Koza (eds.), Morgan Kaufmann: San Francisco, CA, 1997, pp. 286–294.
32. J. P. Rosca and D. H. Ballard, "Discovery of subroutines in genetic programming," in P. J. Angeline and K. E. Kinnear, Jr. (ed.), Advances in Genetic Programming 2, MIT Press: Cambridge, MA, 1996, pp. 177–201.
33. M. Schoenauer, M. Sebag, F. Jouve, B. Lamy, and H. Maitournam, "Evolutionary identification of macro-mechanical models," in Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press: Cambridge, MA, 1996, pp. 467–488.
34. E. V. Siegel, "Competitively evolving decision trees against fixed training cases for natural language processing," in Advances in Genetic Programming, K. E. Kinnear, Jr. (ed.), MIT Press: Cambridge, MA, 1994, Chapter 19, pp. 409–423.
35. T. Soule, J. A. Foster, and J. Dickinson, "Code growth in genetic programming," in The First Genetic Programming Conf. (GP-96), J. P. Koza, et al. (eds.), Morgan Kaufmann: San Francisco, CA, 1996, pp. 215–223.
36. T. Soule and J. A. Foster, "Effects of code growth and parsimony pressure on populations in genetic programming," Evolutionary Computation vol. 6(4), pp. 293–309, 1998.
37. L. Spector, "Simultaneous evolution of programs and their control structures," in Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press: Cambridge, MA, 1996, Chapter 7, pp. 137–154.
38. A. Teller and D. Andre, "Automatically choosing the number of fitness cases: The rational allocation of trials," in The Second Genetic Programming Conf. (GP-97), J. R. Koza, et al. (eds.), Morgan Kaufmann: San Francisco, CA, 1997, pp. 321–328.
39. A. S. Wu and R. K. Lindsay, "Empirical studies of the genetic algorithm with noncoding segments," Evolutionary Computation, vol. 3(2), pp. 121–147, 1996.
40. B.-T. Zhang, "Accelerated learning by active example selection," Int. J. Neural Syst. vol. 5(4) pp. 67–75, 1994.
41. B.-T. Zhang, "A Bayesian framework for evolutionary computation," in The 1999 Congress on Evolutionary Computation (CEC99), Special Session on Theory and Foundations of Evolutionary Computation, IEEE Press, 1999, pp. 722–727.
42. B.-T. Zhang and J.-G. Joung, "Genetic programming with incremental data inheritance," In The 1999 Genetic and Evolutionary Computation Conf. (GECCO-99), W. Banzhaf et al. (eds.), Morgan Kaufmann: San Francisco, CA, 1999, pp. 1217–1224.
43. B.-T. Zhang and H. Mühlenbein, "Genetic programming of minimal neural nets using Occam's razor," in Proc. of 5th Int. Conf. on Genetic Algorithms (ICGA-93), S. Forrest (ed.), Morgan Kaufmann: San Francisco, CA, 1993, pp. 342–349.
44. B.-T. Zhang and H. Mühlenbein, "Balancing accuracy and parsimony in genetic programming," Evolutionary Computation, vol. 3(1) pp. 17–38, 1995.
45. B.-T. Zhang, P. Ohm, and H. Mühlenbein, "Evolutionary induction of sparse neural trees," Evolutionary Computation, Vol. 5(1) pp. 213–236, 1997.