Incremental Evolution in Genetic Programming

Jay F. Winkeler
Electrical and Computer Engineering Dept.
University of California at Santa Barbara
Santa Barbara, CA 93106-9560
E-mail: jay@iplab.ece.ucsb.edu

B. S. Manjunath
Electrical and Computer Engineering Dept.
University of California at Santa Barbara
Santa Barbara, CA 93106-9560
E-mail: manj@ece.ucsb.edu

ABSTRACT

This paper presents a study of different methods of using incremental evolution with genetic programming. Incremental evolution begins with a population already trained for a simpler but related task. No other systematic study of this method seems to be available. Experimental evidence shows the technique provides a dependable means of speeding up the solution of complex problems with genetic programming. A novel approach that protects against poor choices of problem simplifications is proposed, improving performance. Testing performed on tracking problems of multiple stages is analyzed.

1. Introduction

Some researchers in the field of computer learning have expressed skepticism that genetic programming (GP) will scale up from toy problems. This concern comes in part from the difficulty of solving problems with *chicken and egg dilemmas*. A chicken and egg dilemma occurs when the interaction of multiple behaviors aids in the performance of the task despite the fact that none of the behaviors individually increase performance. For instance, no performance advantage exists for a program which writes to memory without a later memory read. Likewise, no advantage is gained by reading from memory unless something useful has been written there. Yet the two behaviors interact to make time-varying actions or responses to time-varying inputs possible. Experience shows that GP can learn memory usage, but what about more complicated activities? What about problems where the chicken and egg dilemmas have more depth or multiple dilemmas exist? Chance rather than predictable progression seems to be required to solve such problems, and chance can be fickle.

Incremental evolution, termed by Harvey et al. (1994), has been used with various evolutionary learning methods to solve complicated problems which may have intrinsic chicken and egg dilemmas. Rather than using a randomly-created initial population, incremental evolution begins with a population already trained for a simpler but related task. A complicated problem is attacked by first training for a simplified version of the problem. Then the difficulty of the problem is incrementally increased by modifications of the fitness function or the training parameters and training continues.

The contributions of this paper are three-fold:

- The existing evolutionary learning technique of incremental evolution is applied to GP. This is shown to increase performance at complicated tasks.
- This paper presents a systematic study of incremental evolution, in particular for a target tracking task. No such study of the technique currently exists.
- A new method for organizing the population when incrementally evolving a problem solution is proposed. This method of *mixed increments* decreases the user's effort in applying incremental evolution without hurting performance.

Incremental evolution is used to discover strategies for tracking a moving object with a camera mounted on a pan-tilt unit. Performance at the tracking task is measured based on tracking error, the average distance of the object from the center of the field of view over a period of time. In one example, ten runs with standard GP achieves an average tracking error of 9.690 pixels. After the same number of program evaluations, ten runs with mixed increments and incremental evolution achieves an average tracking error of 0.075 pixels. The performance improvement is shown to be statistically significant. This paper proposes the use of incremental evolution with the novel modification of mixed increments when attempting to solve complicated problems.

The rest of the paper is organized as follows. Section 2 presents previous work with incremental evolution. Section 3 details the use of incremental evolution and presents a number of possible configurations. Section 4 explains the general parameters for the experiments and how the results are analyzed, without reference to a specific task. Section 5 discusses the target tracking task and Section 6 presents the experimental results. Finally, Section 7 concludes the paper.

2. Previous Work

While a number of researchers have used incremental evolution with various evolutionary learning methods, there seems to be no general comparison of different implementations of the technique.

Harvey et al. (1994) use incremental evolution to train robots to move towards a white triangle on a dark background, but not towards a similarly-sized white rectangle. The robots are first trained to orient themselves to face a large white rectangle. Then the target is changed to a smaller white rectangle and robot pursuit is tested as this target is moved. Finally, training focuses on the actual task of interest, which includes not moving towards the very rectangles the robots have already been trained to pursue. Comparisons made to those trained from scratch find incremental evolution provides more robust solutions.

Nolfi et al. (1994) use incremental evolution to train controllers for robots. Maneuvering physical hardware in real time makes training with robots prohibitively expensive. However, any simulation of the robot uses an approximate model, which simplifies the noise and environment and can influence learning. This research finds significant degradation in performance when solutions trained in simulation are transferred to physical hardware. Training the simulation-evolved controllers for a few generations with the actual robots raises the real-world performance to simulation levels.

Beer and Gallagher (1991) use a technique that may be confused with incremental evolution to evolve controllers for locomotion in simulated hexapod robots. A controller trained from scratch is compared to a controller trained in two steps. In the latter case, the single leg controllers are first trained and then complete motion is trained *with the single-leg parameters held constant*. This alternative method results in solutions which are not as robust as those trained from scratch. In true incremental evolution, the single-leg parameters would continue adapting during the second phase of training.

This paper presents a study of the different methods of incremental evolution described in Nolfi et al. (1994) and Harvey et al. (1994), as well as some alternative methods. Mixed incremental training, which is discussed more formally in the next section, differs significantly from the methods in these previous works. Standard incremental evolution begins with a population already trained on a single task that is simpler than, but related to, the task at hand. In mixed increments, multiple populations trained on a number of simplified tasks are combined to create the starting point. This new technique is shown to be robust against poor choices of simplified tasks.

3. Incremental evolution

Incremental evolution has been used with various evolutionary learning methods to solve complicated problems. Rather than using a randomly-created initial population, incremental evolution begins with a population already trained for a simpler but related task. A complicated problem is attacked by first training for a simplified version of the problem. Then the difficulty of the problem is incrementally increased by modifications of the fitness function or the training parameters. The advantages of this method are as follows.

(1) Chicken-and-egg dilemmas may be reduced by directly solving portions of the problem along the way.

(2) The fitness function for the final problem may be reduced in complexity. Or if the complexity is not reduced, the progression of training makes it less likely that any weaknesses in the final fitness function will be exploited.

(3) Fewer evaluations may be required when training for the simplified problem, making it less costly to reach a given level of performance on the complete problem.

(4) The programs evolved by incremental evolution do not perform worse than those found by training from scratch.

(5) In many cases, the performance for programs trained with incremental evolution is statistically superior to those trained with standard GP.

3.1 An alternative: Fitness-based shaping

An alternative method of solving complicated problems containing chicken-and-egg dilemmas is to guide or shape the learning process through the fitness function. Any steps or tools considered necessary can be directly rewarded. However, this presents some major problems:

- The burden of effort is being shifted to the user who must design a complicated fitness function. One of the main advantages of GP is that it enables the computer to solve problems without being explicitly told what to do. When the user must specify steps along the way, this advantage is diminished.
- The user may incorrectly decide some behaviors are necessary. In the worst case, the behaviors rewarded may in fact make the problem more difficult.
- Complex fitness functions may result in interactions which reward undesirable behavior. The exact effects of complicated fitness functions are difficult to predict. Degenerate solutions may result.
- For a complicated fitness function, it may be necessary to evaluate the program many times to get an accurate measure of fitness. Program evaluations have been shown to dominate the computational cost of GP, so extra evaluations will raise the overall cost.

3.2 Incremental evolution for problem solving

Incremental evolution provides a means for solving complicated problems with evolutionary learning techniques. The complicated problem is first divided into incremental pieces. For many problems, these divisions are intuitive. The pieces may be steps required along the way to completing the entire solution, relatively independent dimensions of the problem, or merely various problem simplifications. The complete problem is solved by training the population of programs incrementally, as the problem definition is gradually altered.

From another point of view, the population for each training run is initialized with a population already trained on a simpler, but related task.

Deciding how the problem should be divided or simplified is an extra task the user has to perform with incremental evolution. In exchange for this extra effort, the user receives a solution for a lower processing cost. In some cases, the extra effort may even make an apparently unsolvable problem solvable.

Consider a tracking problem where the goal is to keep a moving object centered in the camera's field of view. Maybe 10 different velocities and 20 different object positions are deemed necessary for a general solution. Solving this problem with a complete fitness assessment (testing each of the velocities and image positions) requires 200 evaluations per individual per generation. In incremental evolution, the velocity may first be held constant while only the initial position is varied. In this case, 20 evaluations per individual per generation are required for the first increment. Training on the complete problem from scratch for 100 generations requires 20,000 evaluations per individual. Training with incremental evolution for 100 generations on the simplified problem requires 2000 evaluations per individual, or 10 generations of training on the complete program. The population is now partly tuned to the problem at a low computation cost. For such multi-dimensional problems, incremental evolution can save a lot of processing effort.

The incremental alterations to the problem can be implemented through the fitness function or the problem parameters. A complicated fitness function representing the complete task may be progressively simplified to create the initial problem increments. This should avoid some degenerate exploitations of the complicated final fitness function since any weaknesses due to reward interactions will not exist in the simplified versions of the fitness function. Or a simple final fitness function, free of exploitable interactions, may have rewards progressively added to encourage specific behaviors in the initial training runs. In the tracking problem, the fitness function remains unchanged throughout all of the training. The task is incremented by varying the problem parameters. Problems that lend themselves to this type of implementation are the most amenable to incremental evolution.

3.3 Termination criteria

Without considering any particular problem or method of dividing a complicated problem into pieces, there are a number of methods for performing incremental evolution. Termination criteria and organization of the population are controllable parameters which determine the various incremental methods. The expected trade-offs involved in different termination criteria are described in this subsection.

The initial increments of the training can be terminated based on generation count or fitness measure. This differs from standard GP only in that training will then continue with an updated version of the problem. Training for a fixed number of generations requires a good knowledge of the problem. The user may have to observe some initial runs to formulate an intelligent choice of the generation parameter. In fact, a single value may not be appropriate as some problems may exhibit wide variance in the number of generations required to achieve comparable performance.

Training until a performance score is reached is more intuitive than training for a fixed number of generations. This approach ensures some desired level of performance in the population, while limiting the amount of training past that point. Training for too long may lead to overfitting the increment or convergence of the population, either of which may damage the population with respect to the complete problem. The fitness score parameter in these performance-based increments may be specified as a raw number. However, this parameter is more naturally specified as a percentage of the maximum score or a tolerance bound on the maximum score, for a "partial" or "complete" solution, respectively.

3.4 Population organization

The organization of the population falls into two major categories: standard undivided population and *demetic grouping*. Demetic grouping is best thought of and implemented as a generalization of the standard undivided population. Demetic grouping has been used with GP to limit premature convergence of a population and can easily be adapted for use with incremental evolution. In demetic grouping, demes or 'islands' of programs develop fairly independently of each other. Within each deme, the normal methods of selection are used with the standard genetic operations. However, these operations are not performed across demes. One new genetic operation is added when using demetic grouping: *migration*. In migration, an individual is directly copied from one deme to another. Demetic grouping allows GP to search multiple areas of program space in unison, avoid premature convergence, and perform computations in parallel.

A rough version of demetic grouping can be implemented as part of incremental evolution by completely restricting the migration between demes during the initial increment of training. In the later increments, this restriction is relaxed. In the extreme, the populations are completely combined. This algorithm can be implemented with GP code that does not allow demetic grouping by initially training separate populations and then combining them when the problem definition is incremented.

The major advantage of using demetic grouping with incremental evolution is that the population's convergence to a single area of program space during the initial increments of training is limited. Variance in the population enables evolutionary learning to occur, as the differences in the population are amplified by selection. However, the act of training itself can cause population convergence. Training with demetic grouping should counteract any convergence that may occur during the initial training increments, and enable quicker learning in the later increments.

Demetic grouping is normally performed with a single divided population. The rough version of demetic grouping previously described enables some interesting options to be examined. For instance, two demes can each include as many individuals as the final population, if some sort of selection is used to draw individuals from these demes. This method combines the better performance of a large initial population with the lower computation cost of a small population as the programs later grow longer.

Another technique unique to incremental evolution with this form of demetic grouping is called *mixed increments*. In mixed increments, each deme initially attempts to solve a different piece of the problem. This allows multiple portions of the problem to be trained at the same time. In the training that follows, the two populations with different training backgrounds not only compete for supremacy, but adopt and adapt each other's strategies. The hybrids which result are likely to perform better than either of the populations on their own. The user is no longer required to order the increments of the problem for training; only problem division is needed. However, this method may be inappropriate for some methods of dividing the problem.

4. Methodology

This section presents the methodology used for the experiments. The baseline training methods of standard GP are briefly explained, followed by the test of statistical significance used for comparing the baseline methods to those proposed. Subsection 4.2 explains the parameters used for genetic programming. The final subsection presents the configurations of incremental evolution tested in this study.

4.1 Baseline training methods

The different methods of performing incremental evolution discussed in Section 3 are systematically compared to the more standard *training from scratch*. In training from scratch, the population is trained for the entire complex problem starting from a random initialization. This method is expected to achieve adequate solutions and is used as a baseline for comparison.

Since the population is initially trained on pieces of a larger problem, testing should be performed on the programs resulting from this initial training. If programs evolved on a piece of the problem solve the entire problem, then continued training is unnecessary. The strategy of training on only one portion of the problem is tested as *naive dimension reduction*. This method is expected to quickly achieve solutions which perform well on the simplified problem, but do not transfer well to the complete problem.

Due to the small number of samples available, *Student's one-tail t test* (Naiman, 1972) is used in the comparison of results. The null hypothesis that the performance is the same for any two methods is rejected only if there exists sufficient statistical evidence that one method outperforms the other. The test can be performed at various significance levels, these levels indicating the probability that the null hypothesis is rejected in error. The test assumes the samples being compared are drawn from Gaussian distributions, and the significance level indicates the amount of overlap in the tails of the density functions. This paper performs tests at both the 0.05 and 0.01 significance levels to obtain accurate assessments. The assumption of Gaussian distributions for these results may not be completely valid, for most ten-sample sets appear vaguely Gaussian.

4.2 General GP parameters

Table 1 shows the complete tableau for the experiments. *Multi-tree modification* means that all the trees in the individual are modified: the main program and each automatically-defined function (ADF). In crossover, this is performed by selecting a single 'mother' program. A different 'father' program is selected for each of the trees, and the subtree cut from the father replaces the subtree cut from the mother. In the alternative, single-tree modification, random selection is used to choose which tree is altered.

Table 1 Tableau for all experiments

Objective	Center an object in the image
Function set	NEGATE, + , − , × , ÷ , MIN, MAX, MIN3, MAX3, ADD3, MULT3, IF_POSITIVE_ELSE
Terminal set	constants: 0, 0.1, ..., 1.9 and 10, 20, ..., 100. X, the input variable.
Memory	2 variables, with read (R0, R1) and write (W0, W1) access.
ADFs	One with one argument and one with two arguments.
Fitness	See section 5.
Parameters	Population: 5 demes of 100 individuals each. Tournaments of size 5. 10% Mutation rate. 2% Migration rate. 30% Terminal selection rate. 90% multi-tree selection rate.

The experiments are performed with a GP tool which runs steady-state genetic programming. Steady-state GP is a memory-saving method of performing GP. Standard GP requires the storage of two populations: the current population upon which the genetic operations are being performed, and the next generation into which new individuals are being written. Steady-state GP uses only one population. After a new individual is created, the selection technique is used to find a poorly performing program in the current deme or population. The poorly performing program is replaced by the new individual. The genetic operation of reproduction is implicit-

ly carried out, so only crossover, mutation, and migration have to be explicitly performed.

4.3 Incremental evolution parameters

The following methods and parameters are used in the experimental study. The results are compared based not on the generations of training, but on the number of response-producing evaluations per individual. This comparison does not account for the re-use of pre-trained populations, but is a fair comparison for the first attempt at solving a problem.

Problem increments: These simplified problems represent only one dimension of the complete problem. These dimensions are tested with a tighter sampling than the complete problem requires in the hopes of achieving a more robust solution. Since only one dimension is being tested, this sampling represents negligible extra effort in terms of solving the complete problem.

Generation incremental training: Training on one problem increment for 50 generations before training on the complete problem.

Performance incremental training: Training on one problem increment until some performance measure is reached and then training on the complete problem for 50 generations. The performance measure is either seven-eighths of the maximum fitness score, or a score within 5 points tolerance of the maximum score. These are considered "partially" and "completely" solved, respectively.

Demetic incremental training: Training on one problem increment with two independent populations for either 50 generations or until a specified performance level is reached. Each of the two populations trained on the first increment contains 500 individuals. After the initial increment, two populations containing a total of ten demes of 100 individuals each have to be reduced to five demes of 100. In these experiments a new 5 deme population is created by randomly selecting two demes from one of the populations and three demes from the other. A performance-based selection would result in a more fit initial population, but at the cost of extra evaluations.

Mixed incremental training: Training on two independent populations for the two different problem increments. This initial training lasts for 50 generations or until a specified performance level is reached. The population is reduced from ten demes to five as in demetic incremental training.

5. Target tracking task

The previous section defined the genetic programming parameters for solving a general problem. Except for the fitness measure and the function and terminal sets, all of the parameters discussed do not depend on the task itself. This section presents the details of the target tracking task.

The target tracking task attempts to keep a moving object centered in the field of view of a camera mounted on a pan-tilt unit. The camera grabs a 120 by 160 pixel binary image of the scene and locates the center of mass of the object. Segmentation is simplified by using a black rectangular object set on a white background. The image position of center of mass of the object is input to the learning system. The learning system output is then issued as a maneuver command to the pan-tilt unit. The tracking cycle repeats with the next image grab. During training, only a small number of image grabs and associated camera movements are tested.

The distance from the camera to the object ranges from approximately 3 to 30 feet. In both training and testing, the object is constrained to move in the plane of the image. The tracking problem is simplified so that only horizontal (pan) direction tracking is trained.

Goal: Output the command which, when issued to the pan-tilt unit, centers the object in the image. The first move is not scored, so the camera can use this move to estimate the depth or velocity of the object without incurring a penalty. Anytime the object moves out of the image window, however, the position is reported as zero.

Performance is measured as

$$\frac{1}{m(n-1)}\left(\sum_{j=1}^{m}\left(\sum_{i=2}^{n} f(M-d_{ij})\right)\right) \qquad (1)$$

where m is the complete number of different initial conditions as starting position, distance from camera to object, and object velocity are varied; n is the number of camera movements over which the tracking is performed; and d is the distance (in pixels) from the object's position in the image to the center point of the camera's field of view. The function f is defined as

$$f(x) = \begin{cases} x^2, x > 0 \\ 0, otherwise \end{cases} \qquad (2)$$

and M is some maximum distance from the center of the field of view (in pixels), beyond which the reward is zero. This formulation allows fitness scores to fall in the same range for all values of m and n.

For these experiments, $M = 50$ and $n = 4$. The value of m depends on the specific problem increment. The complete problem is divided into three increments: object position in the image, object depth, and object velocity. This divides the problem into three relatively independent dimensions.

Incremental evolution uses multiple training runs to discover strategies for the complete problem. For instance, the initial population may be trained to position the camera when velocity and object depth are held constant. Then the resulting population may be trained for various initial positions and object depths while velocity remains constant. Finally, the resulting population may be trained on the complete problem.

These experiments are performed with three different "complete" tracking tasks. Three different problem increments, defined below, are used for the initial training. In these

experiments, the tracking task is simulated, using parameters obtained from the physical pan-tilt unit.

All of the complete problems examined require some sort of memory usage for a perfect solution. The differences between inputs at different time steps contain the information necessary to track objects at different depths or velocities. The calculation of these differences requires memory, so these tasks have at least one chicken-and-egg problem

Complete problem Position_Depth considers initial image positions $\{-75, -65, \ldots, 75\}$ for an image that is 160 pixels in width. The depth of the object is characterized as $\{1.5, 2, 2.5, \ldots, 9\}$. The object is stationary throughout training, so $m = 256$ for the complete problem.

Complete problem Position_Velocity considers initial image positions $\{-75, -65, \ldots, 75\}$ for an image that is 160 pixels in width. The depth of the object is characterized as 5. The velocity of the image of the object is $\{-60, -50, \ldots, 60\}$ and is held constant over every four-move training. So $m = 208$ for the complete problem.

Complete problem Velocity_Depth considers initial image positions $\{-75, 75\}$ for an image that is 160 pixels in width. The depth of the object is characterized as $\{1.5, 2, 2.5, \ldots, 9\}$. The velocity of the image of the object is $\{-60, -50, \ldots, 60\}$ and is held constant over every four-move training. So $m = 416$ for the complete problem.

Problem increment Position_Only: The initial image positions in the range $\{-75, -70, \ldots, 75\}$ are trained, while the velocity is held constant at zero and the depth parameter is held constant at 5. For problem increment Position_Only, $m = 31$.

Problem increment Depth_Only: The object depths in the range $\{1.5, 1.75, \ldots 9\}$ are trained. The velocity is held constant at zero, and the initial positions of $\{-75, 75\}$ are used. For problem increment Depth_Only, $m = 62$.

Problem increment Velocity_Only: The image-viewed velocity of the object in the range $\{-60, -55, \ldots, 60\}$ is trained. The depth parameter is held constant at 5, and the initial positions of $\{-75, 75\}$ are used. For problem increment Velocity_Only, $m = 62$.

6. Experiments

This section begins with a systematic study of the effects on final performance of two different parameters of incremental evolution: termination criterion for the initial increment and population organization. Training from scratch is compared to naive dimension reduction to show empirically that the dimensions of the problem are not artificial constructs. Finally, programs evolved with incremental evolution are compared to those evolved with standard GP.

The initial training increments are always one of the three described in the previous section. Figure 1 shows the average best-of-generation performance scores for training problem increments Position_Only, Depth_Only and Velocity_Only over the period of training. The average score is computed over ten runs, and the minimum and maximum bounds are also displayed. The x's show the positions where the training reaches seven-eights of the maximum score and the o's show where the score breaks the tolerance range of 5 points of the maximum score. The evaluation count is the number of evaluations per individual in the population, and represents 50 generations of training in all cases.

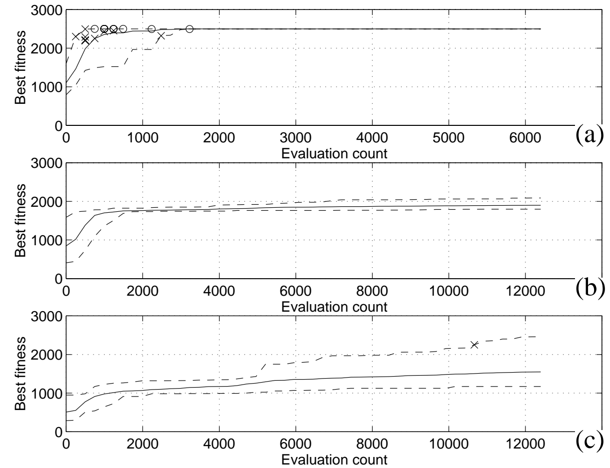

Figure 1 Average best-of-generation performance scores on (a) increment Position_Only, (b) increment Depth_Only, and (c) increment Velocity_Only. The solid curve represents the average score, while the dashed lines represent the minimum and maximum over the ten runs.

6.1 Incremental training termination

As discussed in section 3.3, the two main methods of specifying the criterion for terminating a training increment are generation count and performance threshold. The experiments in this section compare the performance obtained using these two methods for a number of problems.

Figure 2 compares the results for training with demetic grouping on complete problem Position_Depth using different termination criteria for the initial training increment. In all three cases, the first increment of training is for problem increment Position_Only. The problem increment is considered completely solved as soon as any program in the population achieves a performance score within 5 points of the maximum score. A partial solution requires a score that is seven-eighths of the maximum. The maximum score in these experiments is 2500, so a complete solution requires a score of 2495 and a partial solution a score of 2187.

Training with a performance-based termination criterion results in programs that visibly outperform those trained with a generation-count termination criterion. A statistical test rejects the hypothesis of equal performance at the 0.01 significance level. The same rejection rate results when a standard undivided population is used instead of demetic grouping. For this problem, terminating the initial increment when a

specific level of performance is reached improves the performance of the resulting programs.

An examination of Figure 1 can show why this is the case. Training on problem increment Position_Only results in a quick solution, well before 50 generations of training. The subsequent training overfits the programs to the initial training increment and reduces the variance in the population. The choice of 50 generations of training in this case is too large, and the results are obvious.

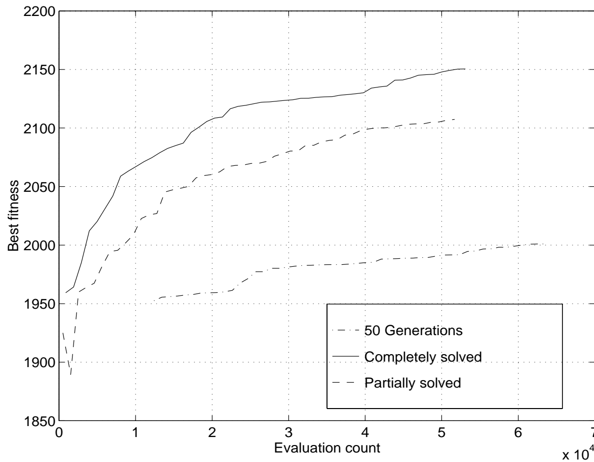

Figure 2 Comparison of termination criteria with complete problem Position_Depth.

Training on problem increment Velocity_Only, on the other hand, results in a partial solution in only one case before the 50th generation. In the other 9 cases, more training is required to achieve even a partial solution. Figure 3 shows the results for training with a standard undivided population on complete problem Position_Velocity when the first increment of training is on problem increment Velocity_Only.

For this problem as well, training with a performance-based termination criterion outperforms a generation-count termination criterion. As before, the statistical test rejects the equal performance hypothesis at the 0.01 significance level. Using the performance-based termination criteria improves the performance of the resulting programs, this time much more dramatically.

Figure 2 might seem to suggest that a population trained on the initial increment until a complete solution is found outperforms one trained until a partial solution is found. In fact, after about 15,000 evaluations have been performed, there exists no statistical evidence to choose one before the other. Only in the initial generations is the advantage of using the complete solution statistically significant. This result, which holds for a standard undivided population as well, may be task and increment dependent.

6.2 Population organization

As discussed in section 3, the two main methods of organizing the population are a standard undivided population and demetic grouping. The rough version of demetic grouping that is tested here completely separates the two populations during the first increment of training and then uses a 2% migration rate when the complete problem is being trained.

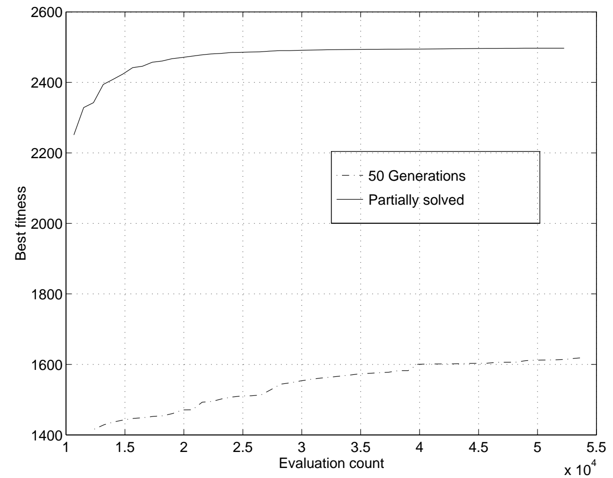

Figure 3 Comparison of termination criteria with complete problem Position_Velocity.

The first test compares the results of training with 2 different methods of organizing the population: standard undivided population and demetic grouping. In both cases, problem increment Depth_Only is trained for 50 generations before the task is incremented to complete problem Position_Depth. In demetic grouping, both demes train on the same problem increment. No statistically significant difference exists between the two organization schemes. The demetic grouping results do not lie far enough above the standard undivided population results to reject the equal performance hypothesis.

Training on problem increment Position_Only with either generation-count or performance-based termination gives the same results. The demetic curve lies above the single population curve, but not with enough difference to reject the hypothesis that the same performance is being observed from both methods.

Figure 4 compares the results for training with a standard undivided population and mixed grouping. In the latter case, the two demes are trained on different increments: Position_Only and Velocity_Only. In all cases, the initial training is terminated at partial solution (when a fitness score of 2187 or higher is reached). Then the two populations are combined, using the algorithm explained in section 4, for training on the complete problem.

The curves in Figure 4 show the main advantage of mixed grouping: the user does not have to select which problem increment to train first. In this case, the choice between increments Position_Only and Velocity_Only as the initial training increment makes a statistically significant difference. Training with increment Velocity_Only significantly outperforms training with increment Position_Only. Mixed grouping performs as well as the better of the two alternatives, but does not require the user to select which training increment to use. Mixed grouping is the safest population

organization, even if the user is familiar with the problem. The solutions evolved with GP can sometimes be surprising, making it difficult to accurately predict what will act effectively as an related task.

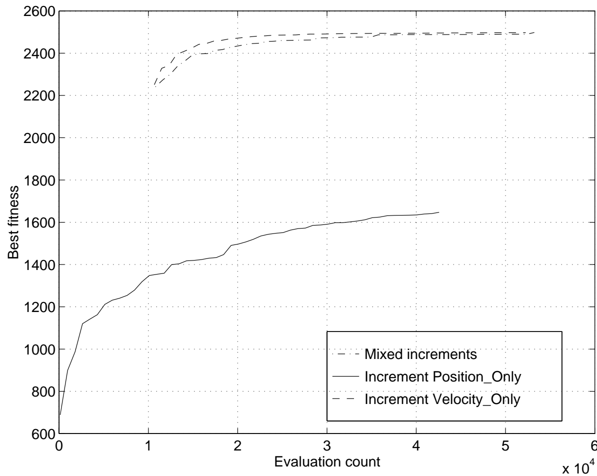

Figure 4 Comparison of population organization with complete problem Position_Velocity.

6.3 Naive dimension reduction

Naive dimension reduction assumes that training for one dimension or increment of the problem will produce a solution to the complete problem. If naive dimension reduction performs well, no further training is necessary. Success with this technique implies the intuitively-selected dimensions of this problem are not actually independent.

Figure 5 compares naive dimension reduction to training from scratch for complete problem Position_Depth. Training from scratch performs better than both cases of naive dimension reduction before many generations of training have elapsed. By the end of training with naive dimension reduction, the results are poor enough to require the rejection of the hypothesis of equal performance. In the training increment Depth_Only case, this rejection is made at the 0.01 significance level. Similar results hold for complete problem Position_Velocity.

Training with naive dimension reduction does not work as well as training on the complete problem. The conclusion can be drawn that the dimensions along which these problems are divided are actually independent. These three dimensions hold different types of information and all need to be trained.

6.4 Incremental evolution experiments

Different configurations of incremental evolution are compared in the previous subsections. These comparisons allow the selection of higher performance algorithms for incremental evolution. This subsection compares these incremental evolution techniques to standard GP.

Table 2 specifies the relationship between performance obtained by incremental evolution methods and performance obtained by training from scratch. In the population organization column, "std" refers to a standard undivided population. The relationship column specifies whether incremental evolution performs better than (>), worse than (<), or equal to (=) training from scratch. The final column indicates the range (in thousands of evaluations per individual) over which the inequality holds before dropping to equality.

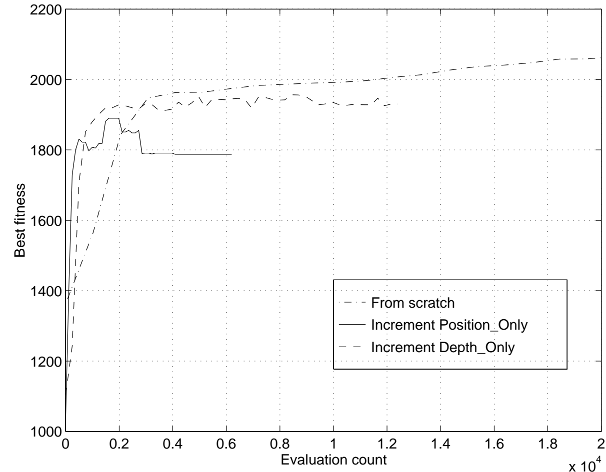

Figure 5 Comparison of naive dimension reduction to training from scratch for complete problem Position_Depth.

In the two cases where standard GP outperforms incremental evolution, the first increment is trained far past the point of solution. This has not just overfit the solution to the training increment. It has allowed the population to converge to a small portion of program space, with devastating effects. In the cases cited as having equivalent performance, the fitness curve of the incrementally evolved solution lies above the curve for standard GP. However, the separation is not statistically significant over these small training sets.

Table 2 Comparison of incremental evolution and training from scratch for complete problem Position_Depth.

Initial Inc.	Term. crit.	Pop. org.	Relat.	Sig. level	Range
Position_Only	solve	deme	>	0.01	[0,15]
Position_Only	part	deme	>	0.01	[0,3]
Position_Only	solve	std	>	0.05	[0,10]
Position_Only	part	std	>	0.05	[0,7]
Depth_Only	gen50	deme	=		full
Depth_Only	gen50	std	=		full
Position_Only	gen50	deme	<	0.01	full
Position_Only	gen50	std	<	0.01	full

Table 3 displays results in terms of tracking error (in pixels) for the complete problems. The training set and two different interpolation sets are tested. "Int. A" denotes values trained in the initial increment but not in the complete problem, while "Int. B" denotes values which have never been

trained. The letter in parentheses in the configuration column specifies the initial training increment.

The solutions trained from scratch are shown to be so robust to interpolation values that no program can do better, only as good. Performance on interpolation values is actually better than the performance on the training values. Most of the solutions trained with incremental evolution show a similar trend. A few, such as the final entry of the table, however, do show a decrease in performance when working with untrained values. These less robust solutions do not occur in most cases, and performance does remain above that evolved from scratch.

Table 3 Comparison of tracking errors for the training set and two interpolation sets.

Problem	Configuration	Tracking error (pixels)		
		train	int. A	int. B
Position_ Depth	from scratch	3.886	3.171	3.187
	gen50 (Depth), deme	3.837	3.246	3.229
	part (Pos.), std	3.535	2.936	2.920
Position_ Velocity	from scratch	9.690	9.166	9.566
	part (Vel.), std	0.075	0.095	0.755
	part (Pos.), std	9.839	9.154	9.851
Depth_ Velocity	from scratch	22.551	21.130	21.760
	part (Vel.), std	16.459	18.592	19.292

7. Conclusions

Training with incremental evolution reduces the number of evaluations necessary to reach a given level of performance with genetic programming. In the experiments persented in this paper, the training data performance for programs trained with incremental evolution is often statistically superior to those trained with standard GP. Most of the programs evolved by incremental evolution are as robust to untrained data as those found by training from scratch. These experiments have found no cases where training with intuitively selected increments does harm, as long as a reasonable termination criteria is used.

The best termination criteria is based on reaching some percentage of the maximum possible fitness score. Training on the initial increments for not long enough, or especially too long, is shown to degrade performance when the problem is incremented. The precise percentage seems to matter little, but this may be dependent on the task and the choice of problem increments.

Mixed incremental training, introduced in this paper, is the safest choice for any user, especially one unfamiliar with GP and the particular problem being trained. For some problems, intuitive simplifications do not seem to capture the essence of the problem well enough to act as 'related' tasks. However, as long as the population retains some variance, it should eventually be able to recover from a poor choice of incremental training.

In section 3, four problems were mentioned with using the fitness function to reward specific steps considered necessary: additional user effort, poor decisions on reward, fitness function interactions, and more evaluations. Incremental evolution does not solve these problems, but reduces them:

- The user must do more work than with standard GP: the problem must be divided and the training increments implemented. However, mixed incremental training lessens the effort by reducing the cost of a poor choice of problem increment.
- The user can still make poor reward or problem division decisions, but mixed incremental training keeps these from affecting the entire result.
- Incremental training can be formulated to avoid degenerate solutions due to fitness interactions. Increments implemented in problem parameters do not require complicated fitness functions.
- Incremental training requires fewer costly evaluations to achieve the same level of performance as standard GP.

The solutions found in this research have overcome at least one chicken-and-egg dilemma, memory usage, without being explicitly rewarded. For difficult problems where efficient solutions to chicken-and-egg problems are required, incremental evolution with mixed increments is worth using. In the best cases, performance is dramatically increased.

Acknowledgments

This research partly supported by NSF grant NSFIRI-9704785.

Bibliography

Beer, R.D. and Gallagher, J.C. 1995. Evolving Dynamical Neural Networks for Adaptive Behaviour. *Adaptive Behaviour.* 1 (1) pp. 91-122.

Harvey, I., Husbands, P., and Cliff, D. 1994. Seeing the Light: Artificial Evolution; Real Vision. *From Animals to Animats 3: Proc. 3rd Int. Conf. on Simulation of Adaptive Behavior.* 392-401.

Naiman, A., Rosenfeld, R., and Zirkel, G. 1972. *Understanding Statistics*. New York: McGraw-Hill Book Company.

Nolfi, S., Floreano, D., Miglino, O., and Mondada, F. 1994. How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robots. *Proc. Artificial Life IV.* 190-197.