Virtual Ramping of Genetic Programming Populations

Thomas Fernandez

Florida Atlantic University, Department of Computer Science and Engineering,
777 Glades Road, Boca Raton, Florida, 33431-0991
tom@cse.fau.edu
www.cse.fau.edu/∼thomas

Abstract. Genetic Programming often uses excessive computational resources because the *population size* and the *maximum number of generations per run* are not optimized. We have developed a technique called Virtual Ramping which eliminates the need to find an optimal value for these parameters and allows GP to be applied effectively to more difficult problems. Virtual Ramping continuously increases the size of a virtual population and the number of generations without requiring additional system memory. It also allows sub-populations to evolve independently which reduces premature convergence on suboptimal solutions. Empirical experiments with problems of varying difficulty, confirm these benefits of Virtual Ramping.

1 Introduction

Genetic Programming(GP) is expected to be computationally intensive. Unfortunately it often uses far more resources than necessary because two parameters, *population size* and *maximum generations per run*, are rarely optimized. These two parameters are perhaps the most important[1] because they set the space and time requirements for a GP run. It is possible to determine the optimal values for these parameters[2], but this is rarely done with real world problems because the determination of these optimal values is done empirically and is too computationally intensive. Research in Genetic Algorithms has shown that optimal population sizes will vary with the problem complexity [3] and it is reasonable to expect that optimal population sizes in GP will also depend on the problem being solved.

In previous experiments the difficulty in determining optimal values has forced us to just guess at the appropriate *population size* and *maximum generations per run*. Sometimes the selection of these parameters has been based on values used by other researchers with different problems. At other times, we would set the *population size* to some large value based on the amount of memory available. Our experience has shown us that regardless of the theoretical optimal population size, if it is set too large then virtual memory is required and this will dramatically slow down GP. Research has shown that in general larger populations are better [4], but computational effort was not considered in this conclusion, and it did not address whether better results could be achieved for the same effort by using smaller populations with more runs or more generations per run.

In our previous work we also had no methodology for setting the *maximum number of generations per run*, so we would set it to a large value, based on the limit of the amount of computer time that we had available to expend on a problem.

In his first book on GP Koza explains that more runs with fewer generations per run can be more efficient but again the computational effort involved in determining the optimal values is prohibitive for real world problems[2]. Only with toy problems is it it feasible to solve the problem hundreds of times to determine optimal parameters values.

To address the GP problem of finding optimal *population sizes* and *maximum numbers of generations per run* we have developed a technique called Virtual Ramping (VR). VR eliminates the need to predetermine these parameters by continuously increasing a virtual population size and the number of generations until GP has sufficient space and time resources to solve a given problem. We refer to our technique of dynamically increasing a GP population size as Virtual Ramping because it does not require an increase in system memory or the use of virtual memory as the virtual population size grows. The virtual population is represented by an actual population of individuals which is fixed in size.

VR also allows parts of the virtual population to evolve independently of each other. Researchers who work with endangered species understand that if the diversity in a population drops below a certain level, the populations viability can be restored by introducing new individuals into the population[5]. This benefit has been reported when sub-populations are maintained separately during parallel implementations of GP[6]. Maintaining separate sub-populations prevents an individual that is a local maxima, from saturating the entire population with it's genetic material and impeding the search for better solutions. Other research shows that separate sub-populations in parallel GP systems reduces code bloat[7].

In this research we experimentally determine that GP with VR evolves better solutions than GP without VR when both are given the same amount of computational resources. By better solutions we mean that the models evolved using VR have a higher accuracy when classifying observations outside of the training data.

2 The Test System

The Genetic Programming System we have written for these experiments is called NUGP. It is a classical Koza type GP system designed as an experimental test bed for new GP techniques. It has an object oriented design where the GP individuals are based on the *Strategy* pattern [8]. Its most important class is a singleton called *Experiment*. NUGP includes a Statistical Analyzer that compares test scores to see if there are significant statistical differences between sets of GP runs[9].

3 Fair Allocation of Computational Resources

In order to allow a fair comparison of different GP systems we need to be certain that each is given the same amount of computational resources. Often computational effort in GP is measured in generations, but this may not be accurate. Generations usually take increasing amounts of time as GP progresses. This is illustrated by the graph in Figure 1 which shows the average time per generation in one of our typical GP runs (without VR) over a 5 minute period. The latter generations are taking 6 to 7 times as long as the first generation.

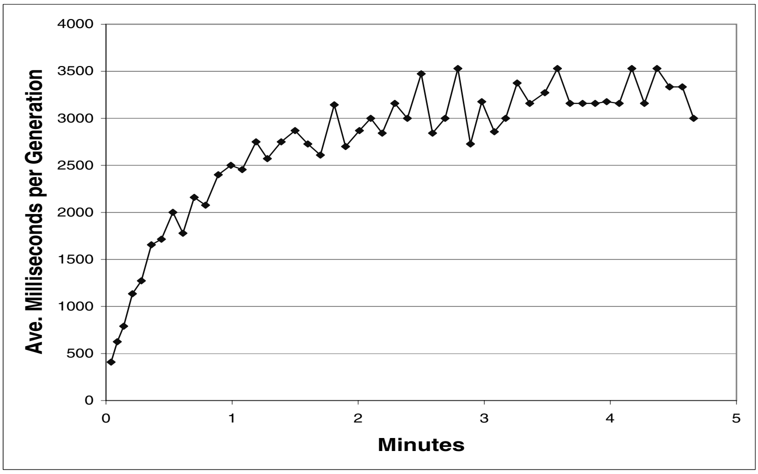

Fig. 1. Average milliseconds per generation as GP runs progress in time

This increase is apparently caused by code bloat[10] as illustrated in the graph in Figure 2, which shows that the increase in *time per generation* corresponds closely to the increase in the average tree size (with a coefficient of linear correlation equal to 0.97).

At best counting the number of generations per run is an uneven measure of computational effort in GP and at worst it is an unfair one. Since GP with VR tends to generate smaller S-expression trees it uses less computational resources than GP without VR when they are run for the same number of generations.[1]

[1] Strictly speaking NUGP does not have generations. It runs asynchronously. We do however still have an analogous term *Round* which is the time during which an asynchronous system will allow n individuals to mate where n is the size of the population. In this paper we will refer to a *Round* as a Generation.

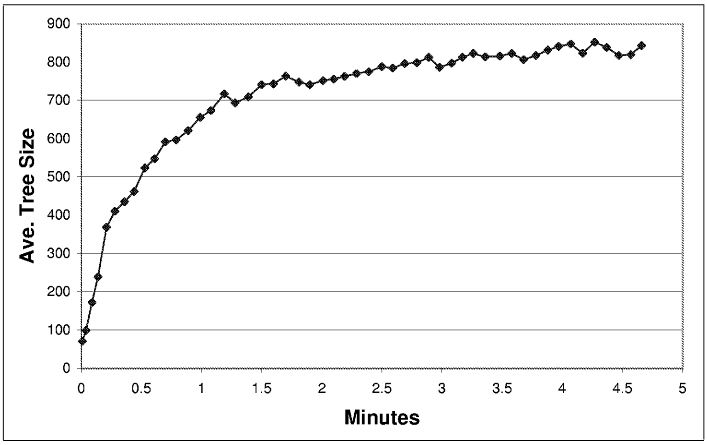

Fig. 2. Average S-expression tree size as GP runs progress in time

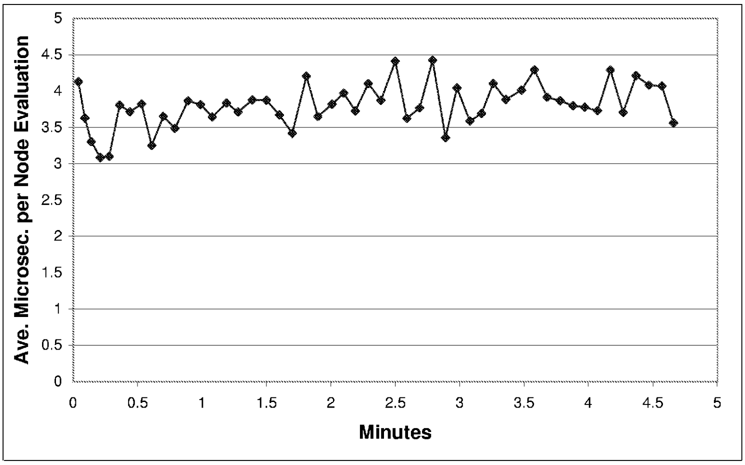

Fig. 3. Average microseconds per S-expression node evaluation as GP runs progress in time

Execution time has been used as a measure of GP system resources [11]. This might work on some platforms but NUGP runs on multiple machines with different configurations that do not have a real time operating system. One of our goals for NUGP was to be able to perform GP runs on different systems and still be able to compare the performance objectively.

We have chosen the *the number of S-expressions nodes evaluated* as our measure of computational effort. We have chosen it because it is independent of hardware performance and software implementation. In the graph in Figure 3 we see that the average number of microseconds per node evaluation is reasonably consistent throughout a GP run. This confirms that the *number of S-expressions evaluated* is a good unit of measure for allocating resources in GP research. Since this number is fairly large we use the term MNE to stand for Million Nodes Evaluated.

4 Methodology

To understand Virtual Ramping we will first consider another hypothetical technique, which we will refer to simply as Ramping. Ramping cannot actually be implemented because its memory requirements grow exponentially. We will then examine Virtual Ramping and see that it provides the benefits of Ramping while maintaining a constant memory requirement. The flow chart for the Ramping process is shown in Figure 4. Ramping would be implemented as follows. A variable s is set to the initial population size. Typically it would be started at a small value such as ten. A variable g is set to an initial number of generations. This would also be a small value such as one. A GP population referred to as the *saved population* or P_{saved} would be generated with s individuals and run for g generations.

Now the main loop of the Ramping algorithm will begin. Another population called P is generated with a population size of s and allowed to run for g generations. Then population P is merged into population P_{saved}, which will double it in size. Population P_{saved} is allowed to run for g more generations. At this time the variables s and g will both be doubled. Then the best individual in P_{saved} is examined to see if it is a satisfactory solution and in this case process is done and has been successful. If the problem is not solved and memory or time resources are not exhausted then the loop will continue to run.

Since the values of s and g are doubled, each time through the loop, the population P_{saved} will continually double in size and will keep running for twice as many generations. In this way Ramping will continuously double the memory and time resources used, until a solution is found or the resources are exhausted.

The advantage of Ramping is clear. The GP practitioner does not have to determine or guess the required *population size* and the *maximum number of generations per run* needed to solve a problem. Less difficult problems will be solved quickly with small populations running for fewer generations. Harder problems will be solved (hopefully) as the population size and the number of generations are increased sufficiently. The disadvantage of Ramping is also clear. The constant doubling of the population size will quickly exhaust all available computational resources.

How can we modify this technique so that the memory requirements do not grow exponentially? The answer is a modified version of Ramping, which we call Virtual Ramping or VR. At each point where Ramping would double the population size, VR also discards half the population using a tournament selection to ensure that the less fit individuals are more likely to be discarded.

In this way VR maintains a constant memory requirement. Since a tournament selection is used when cutting the population in half, the surviving half

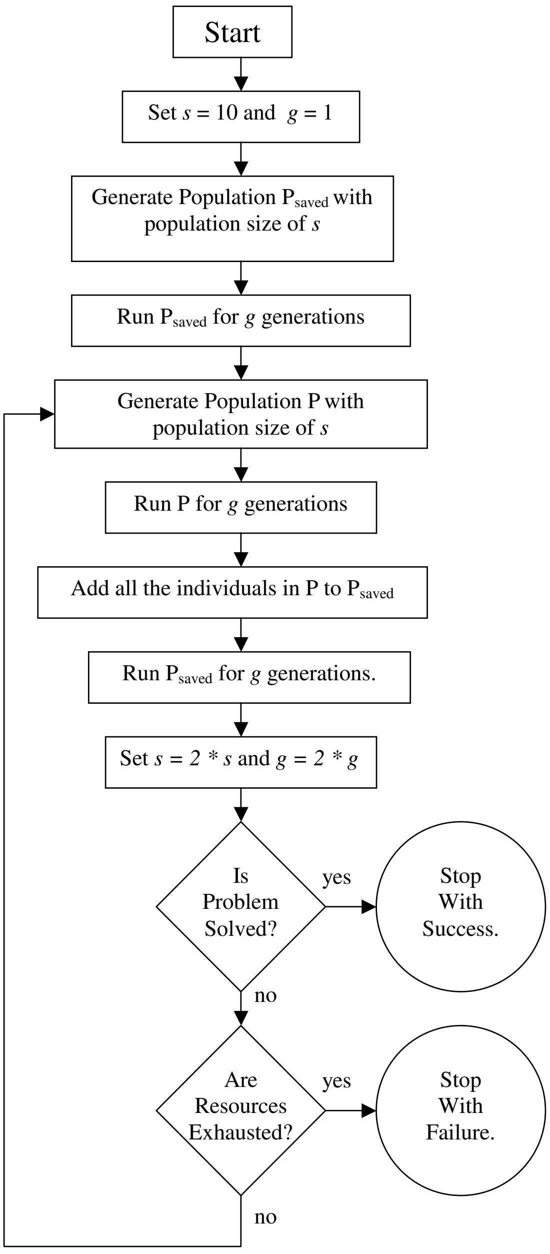
Start
Set s = 10 and g = 1
Generate Population P_{saved} with population size of s
Run P_{saved} for g generations
Generate Population P with population size of s
Run P for g generations
Add all the individuals in P to P_{saved}
Run P_{saved} for g generations.
Set $s = 2 * s$ and $g = 2 * g$
Is Problem Solved?
yes
Stop With Success.
no
Are Resources Exhausted?
yes
Stop With Failure.
no

Fig. 4. Flow Chart for Ramping

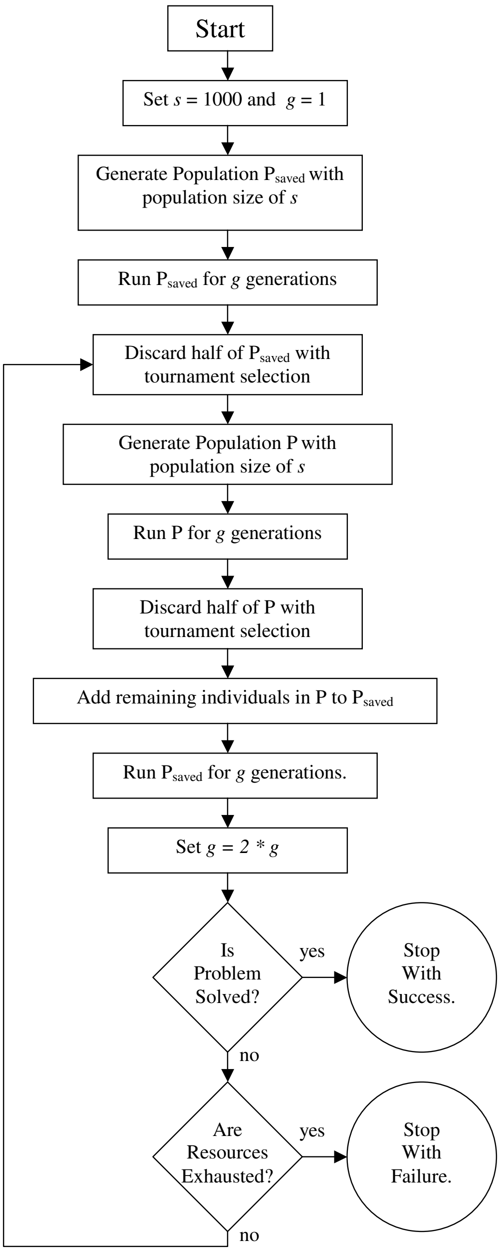
Start
Set s = 1000 and g = 1
Generate Population P_{saved} with population size of s
Run P_{saved} for g generations
Discard half of P_{saved} with tournament selection
Generate Population P with population size of s
Run P for g generations
Discard half of P with tournament selection
Add remaining individuals in P to P_{saved}
Run P_{saved} for g generations.
Set $g = 2 * g$
Is Problem Solved?
yes
Stop With Success.
no
Are Resources Exhausted?
yes
Stop With Failure.
no

Fig. 5. Flow Chart for Virtual Ramping

should retain the majority of the population's "good" genetic material and thus keep most of the benefit of Ramping without the exponential increase in memory requirements.

The flow chart for VR, Figure 5 shows the details of how this works. It is essentially the same as Ramping with a few minor but important changes. At the end of the loop the value of g is doubled but the value of s does not change. Before the population P is merged into P_{saved} half of the individuals are discarded from both populations using a tournament selection in an effort to keep the best individuals. These reductions ensure that the size of population P_{saved} will never exceed the value of s. Since the value of s remains constant it is started out at a much larger value such as 1000.

5 The Experiments

We now ask, does VR benefit the GP process? It eliminates the need to determine the *population size* and the *maximum number of generations per run* but does it generate solutions that make more accurate classification of test observations that were not used during training? To answer this question we have conducted some experiments. VR has been tested using three problems of different difficulty which we refer to as *Easy*, *Medium*, and *Hard*. All of the problems are symbolic regression problems with a set of independent variables and a dependant variable, which was derived from a predetermined formula. For each problem there were three sets of 30 observations. There was a training set, which was used by GP for training, and there were two tests sets referred to as *Test1* and *Test2*. GP was trained using the training set and a fitness function described in the following formula:

$$fitness = w_e \left(\frac{\sum\limits_{i=1}^{n} (2^{E_i} - 1)}{n} \right) . \tag{1}$$

E_i is the prediction error for observation i defined as

$$E_i = |y_i - \hat{y}_i| . \tag{2}$$

Where y_i and $\hat{y}_i$ are the value of dependent variable value and the prediction of the dependent variable for observation i. In this case n is 30 because there are 30 observations in the training data set.

The use of 2^{E_i} in formula (1) makes it an exponential fitness function, which is a means of directing GP to concentrate on the largest errors. The reason for subtracting one from the exponential error term is so that zero will signify no error.

After using the exponential fitness function to drive the GP process the best individuals from each GP run are tested using the two sets of test observations. The evaluation function for the test sets consists of the percentage of the observations, which are misclassified in one of the three categories: *Less than zero*, *Close to zero*, and *Greater than zero*.[2] An observation will be considered correctly classified if both the dependant variable and the GP model prediction are both greater than zero, both less than zero, or both within a given threshold of zero, i.e., both close to zero.

For the *Medium* and *Hard* problems the threshold for being close to zero was set to be between -2 and 2. The *Easy* problem was so easy that at first GP always evolved models that were 100% accurate at classification with or without VR. The ability to do this is a credit to GP, but also makes the *Easy* problem useless for the purpose of evaluating the benefit of VR. In order to make the *Easy* problem more difficult the threshold for being close to zero was set to being exactly zero. Also where GP was applied to the *Easy* problem the actual population size was reduced from 1000 to 100. After making these changes the *Easy* problem was tackled with and without VR and we were then able to determine that the system with VR produced better results.

For each of the three problems an experiment was run with one hundred GP runs with VR and another one hundred runs without. During each generation 98% of the new individuals were created through reproduction. Individuals were selected for mating with a tournament size of two. Mutation was applied to 2% of the individuals. Individuals were randomly selected for mutation with a uniform distribution. S-expression trees were limited to 1024 nodes. Initial populations were generated with sparse and bushy trees with depths between 3 and 6 levels. In each generation Numeric Mutation was applied to 10% of the numeric constants in 10% of the population. Numeric Mutation replaces the constants with a randomly generated value using a normal distribution around the previous value and a standard deviation of an individual's fitness score divided by ten[12]. All of these parameters were held constant for all runs. All runs in the experiment with the *Easy* problem were allowed to continue until 100 million S-expression nodes had been evaluated. In the experiment with the *Medium* and *Hard*problems, runs were executed until one billion S-expression nodes were evaluated.

The GP function set includes addition, subtraction, multiplication, protected division, and a function called IFGTZ (if greater than zero). IFGTZ has three arguments. If the first argument is greater than zero then it will return the second argument, otherwise it returns the third argument. The terminal set included the independent variables and the random ephemeral constant.

In each experiment the only difference between the two sets of runs was that VR was used with one set and not used in the other.

The training and the two test data sets for each of the three problems were synthetically generated. For the *Easy* problem each observation consisted of two independent variables, which were selected at random between -2 and 2 with a uniform distribution. We will refer to these two values as x_1 and x_2. The dependent value y was calculated using the formula

$$y = (3.456x_1) + (6.543/x_2)\ . \tag{3}$$

[2] all of the data has been normalized to have an mean of zero.

For the *Medium* problem each observation consisted of five independent variables, which were also selected between −2 and 2 with a uniform distribution. The independent variables are referred to as x_1,x_2,x_3,x_4 and x_5. The dependent value y was calculated for each observation as

$$y = (x_1 x_2) + ((3x_3^2 + 5x_4)/x_5) \ . \tag{4}$$

The *Hard* problem was generated in the same way as the *Medium* problem with two important exceptions. After calculating the dependent variable, each of the independent variables values was modified by replacing it with a randomly generated value selected between plus or minus 10% of its value using a uniform distribution. This random component is a simulated measurement error added to make the problem harder for GP and more like a real world problem. Also five additional independent variables were added, also randomly generated like the original five independent variables, but these were not used to generate the dependent variable values. There were added to challenge GP's ability to do model selection and determine which independent variables should be used when building models.

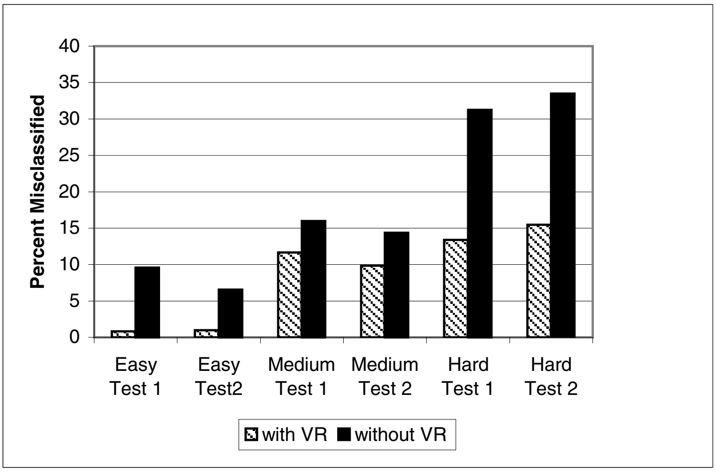

Fig. 6. Average percent misclassified in test data sets with and without Virtual Ramping

6 The Experimental Results

For all three problems 100 GP runs were made with VR and another 100 GP runs were made without VR. As shown in Figure 6, for all three cases the average percent misclassified (in both test sets) was smaller with VR.

To understand the significance of this we must also consider the sample variances and calculate the z statistic. This data is summarized in Table 1. In all

Table 1. Statistics for the three problems with and without Virtual Ramping. (All samples consist of 100 runs.)

	Easy Problem		**Med. Problem**		**Hard Problem**	
With VR	Test 1	Test 2	Test 1	Test 2	Test 1	Test 2
Ave. Misclassified	0.00833	0.00967	0.11667	0.09867	0.13367	0.15467
Variance	0.00142	0.00108	0.00946	0.00666	0.01357	0.01442
Without VR						
Ave. Misclassified	0.09567	0.06567	0.15967	0.14367	0.31233	0.33500
Variance	0.01429	0.00715	0.00676	0.00617	0.02875	0.02690
z statistic	-6.9670	-6.1720	-3.3759	-3.9727	-8.6855	-8.8713

Table 2. Statistics for the three problems with and without Virtual Ramping in the repeated experiments. (All samples consist of 100 runs.)

	Easy Problem		**Med. Problem**		**Hard Problem**	
With VR	Test 1	Test 2	Test 1	Test 2	Test 1	Test 2
Ave. Misclassified	0.00933	0.01067	0.11367	0.09567	0.11700	0.14867
Variance	0.00103	0.00065	0.00657	0.00432	0.01011	0.01435
Without VR						
Ave. Misclassified	0.08933	0.06600	0.16433	0.13133	0.30167	0.31733
Variance	0.01066	0.00563	0.00600	0.00615	0.02770	0.02832
z statistic	-7.3980	-6.9814	-4.5177	-3.4856	-9.4957	-8.1653

cases the absolute value of the z statistic is greater than 2.576 (required for 99% confidence with a two tailed test). We conclude (with a 99% certainty)[13] that for all three problems VR reduces misclassification in test data sets. All three experiments were repeated with similar results shown in Table 2.

7 Conclusions

We have run experiments with three different problems, which we have named *Easy*, *Medium*, and *Hard*. In each experiment we have concluded, with a 99% confidence, that Virtual Ramping improves GP's ability to classify observations outside of the training set. We found that this observation was true using two separate sets of test data for each of the three problems. To further reinforce our conclusion, the entire experiment was repeated with similar results.

The technique of Virtual Ramping could be applied to other forms of Evolutionary Computation. We suspect that it will provide similar benefit but experiments are needed to confirm this.

References

1. R. Feldt and P. Nordin. Using factorial experiments to evaluate the effect of genetic programming parameters. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP 2000, Edinburgh, UK, April 15-16, 2000*, volume 1802 of *LNCS*, pages 271–282. Springer-Verlag, 2000.
2. J. Koza. *Genetic programming: on the programming of computers by means of natural selection.* MIT Press, 1992.
3. J.T. Alander. On optimal population size of genetic algorithms. In *CompEuro 1992: Computer Systems and Software Engineering Proceedings*, pages 65–70, 4–8 May 1992.
4. K. M. Dill, J. H. Herzog, and M. A. Perkowski. Genetic programming and its applications to the synthesis of digital logic. In *Communications, Computers and Signal Processing, PACRIM 1997*, volume 2, pages 823–826, Victoria, BC, Canada, 20-22 August 1997.
5. K. Nowell and P. Jackson. *Wild Cats: Status Survey and Conservation Action Plan.* IUCN Publications Services, 1996.
6. Makoto Iwashita and Hitoshi Iba. Island model GP with immigrants aging and depth-dependent crossover. In David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 267–272. IEEE Press, 2002.
7. G. Galeano, F. Fernandez, M. Tomassini, and L. Vanneschi. Studying the influence of synchronous and asynchronous parallel GP on programs' length evolution. In *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002.*
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
9. R. Jain. *The Art of Computer System Performance Analysis.* Wiley, 1991.
10. W. B. Langdon. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.
11. N. Tokui and H. Iba. Empirical and statistical analysis of genetic programming with linear genome. In *System, Man and Cybernetics: IEEE International Conference Proceedings, Tokyo, Japan, 1999*, pages 610–615 vol.3. IEEE Press, 1999.
12. T. Fernandez and M. Evett. Numeric mutation as an improvement to symbolic regression in genetic programming. In *Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming, San Diego, California, USA, March 25-27, 1998*, 1998.
13. W. Mendenhall and L. Ott. *Understanding Statistics.* Duxbury Press, 1980.