Methods for Evolving Robust Programs

Liviu Panait and Sean Luke

Department of Computer Science, George Mason University
4400 University Drive MSN 4A5, Fairfax, VA 22030, USA
{lpanait,sean}@cs.gmu.edu
http://www.cs.gmu.edu/~eclab

Abstract. Many evolutionary computation search spaces require fitness assessment through the sampling of and generalization over a large set of possible cases as input. Such spaces seem particularly apropos to Genetic Programming, which notionally searches for computer algorithms and functions. Most existing research in this area uses ad-hoc approaches to the sampling task, guided more by intuition than understanding. In this initial investigation, we compare six approaches to sampling large training case sets in the context of genetic programming representations. These approaches include fixed and random samples, and adaptive methods such as coevolution or fitness sharing. Our results suggest that certain domain features may lead to the preference of one approach to generalization over others. In particular, coevolution methods are strongly domain-dependent. We conclude the paper with suggestions for further investigations to shed more light onto how one might adjust fitness assessment to make various methods more effective.

1 Introduction

The bulk of evolutionary computation has been applied to non-stochastic problems with a finite set of inputs. Because the problem input space is fixed, the quality of a candidate solution can often be determined precisely, and often rapidly. More formally, given a set A of candidate problem solutions, much of evolutionary computation is typically trying to maximize some function f over a single fixed context c : $\text{Arg Max}_{a \in A} f(a, c)$.

There are important and notable exceptions to this general trend in EC. Interestingly, genetic programming has not been one of them. We say that this is interesting because genetic programming's notional goal is the development of *computer programs* or *algorithms* which are human competitive. But perhaps because of the difficulty of the search space, much of the GP community has focused instead on simple problems with little computational complexity, and thus needing only a finite, small input context. Even problems like Symbolic Regression, which technically have an infinite input space, are reduced to a fixed set of samples. Nonetheless, as computer power increases we expect to see the community more and more trying to tackle bigger computational challenges, demanding fixed and variable amounts of internal state, iteration, and recursion.

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2724, pp. 1740–1751, 2003.

Often, such more “challenging” algorithmic problems range over an *infeasibly large* set of inputs. By “infeasibly” we mean that the set is so large that there is no way that the algorithm may be exhaustively tested on every possible input. In fact many, if not most, common algorithms operate over an *infinite* sized set of inputs, and inputs in these sets often differ in size or difficulty. Compare, for example, a function which sorts a single vector of predefined numbers to a computer algorithm which can sort any vector of any size and content. More formally we might describe these latter ones as optimization problems: Arg $\mathrm{Max}_{a \in A}\, f(a, \langle c_0, c_1, ..., c_\infty \rangle)$. In many cases this optimization may be described as a summation: Arg $\mathrm{Max}_{a \in A} \sum_{i=0}^{\infty} f(a, c_i)$ This class of problems is challenging because there is no way to prove through empirical exhaustion that such an algorithm is correct or optimal, simply because the input space is too large.

Certain “non-algorithms” can fall into this class as well. For example, real-valued feed-forward systems such as neural networks or symbolic regression trees may operate over an infinite set of numbers. Additionally, some stateless functions are intended to be repeatedly pulsed to iteratively manipulate an external world state: a soccer robot might have a simple set of boolean classification functions to be tested against an infinite number of possible opponent contexts.

As EC has been applied to problems in this class, various techniques for dealing with generalization over the input space have been proposed and tested. One such approach, *coevolution* has also attracted some theoretical attention. In this paper we will discuss and compare several common approaches to doing evolutionary computation in the face of large sets of inputs, and will cast them in the context of genetic programming.

We admit up front two ways in which our methodology will seem odd given the justifications described earlier. First, none of the problem domains we used actually has iteration, recursion, or internal state. We chose them instead because the goal in the study is to compare methods for large input spaces, rather than introduce new domains. As such we felt it more useful to use common and readily implementable problem domains relevant to the genetic programming literature. Second, certain problem domains tested (such as 4-parity) do not have large input spaces: however these problems do have a property central to the experiment, namely that a small sample of the input space does not provide many clues about the input space as a whole.

2 Robustness

In this paper we adopt the term *robust* and presume, as did [1], that it is synonymous with *generalizable*. In the machine learning community both terms imply the ability of a hypothesis induced from a set of examples (in our case, the learned genetic program) to adequately model the entire universe of exemplars (inputs). In [2], the notion of robustness is also attached to the ability to continue to perform well despite mutations in the evolved program code (distinguishing such “genotypic robustness” from generalizability or “phenotypic robustness”). We will not consider this issue in our study.

How does one go about searching for robustness? If possible, one may begin by attempting to reduce the input space to a set of "prototypical inputs" which can be proven to be a sufficient set to learn on. For example, when evolving a sorting network of size n it is not necessary to sort vectors of all possible integers; instead it is sufficient to sort all vectors consisting of only zeros and ones [3]. In many cases, however, this reduction is not possible, not obvious, or insufficient to reduce the input space to manageable sizes. Beyond this, a learning system has no choice but to sample the space.

The space may be sampled in a variety of ways. The obvious approach is to establish a fixed initial random sample which is repeatedly presented until the system has learned it. A naive variation is to fix the sample to a few input cases and hope for the best: this is the approach common in Symbolic Regression, for example. One interesting statistical issue not considered in this paper is the size of the sample: an extreme approach is to use one input case and evaluate as many candidate solutions as possible. At the other extreme, excessively large samples may waste evaluation resources that could be better spent on testing other potential solutions. We realize that such results may be sensitive to the sample size and plan to investigate this aspect in the future. For this work, we decided to fix the sample size to a relatively small fraction of the input space as we think this will be representative of typical problems.

Another approach is to randomly and uniformly resample the space at each presentation. This has the benefit of hindering convergence to a predefined set of exemplars, but a constantly changing input sample can also prevent the candidate solutions from having any search gradient. Sampling may also be done adaptively. There are two popular adaptive methods in the literature. *Sharing* methods discount the value of a given input based on how easy it is for the population as a whole to solve it. *Coevolutionary* methods evolve inputs along with the candidate solutions: the fitness of an input is based on how many candidate solutions it stumps.

It has been often the case that robustness was a desirable property for the end-result of the search process. This property comes however with overwhelming drawbacks. First, there is little understanding on how to proceed about searching for individuals that exhibit this feature. Second, because a large number of input cases may be necessary, the search process may require an increased computational time. Third, it is not trivial to test the presence of the property in the results of the evolutionary process (especially for prohibitively large numbers of possible inputs). In this paper, we plan to shed some light on the first and last concerns, while also investigating whether small samples of input cases can provide enough information for robust programs.

[4] presents a good survey of existing work, as well as an excellent study on evolving robust programs for the Artificial Ant problem by modifying the ant's food trails and creating new ones. [3] coevolves sorting networks with training sets and obtains a remarkable success in a difficult domain. [5] reports obtaining more robust programs when using coevolution, but, as we will see later in the experimental section, this conclusion is strongly domain-dependent. [6] suggests that adding the inverse size to the fitness increases the generality of evolved

programs, while [7] states that less robust results are obtained when a preference for smaller programs biases the GP system; [8] further investigates the relation between evolved program size, generality, and modularity. [9] presents a method for producing highly parsimonious DAG-based boolean programs which can generalize using only a small set of the subfunctions and variables in a problem. Adding noise to the inputs [10] or using multiple fitness sets, augmentation, and refinement [1] are other approaches to improving robustness of evolved programs.

3 Evolving Robust Programs

We selected six methods as candidates for our study. All six methods are either widely used or were reported to have yielded robust results. In our description of the methods below, we define $\mathrm{Err}(p, c)$ to be the error of a program p on the training case c.

Fixed-Random-Initial (FRI) randomly selects a set $c_1, ..., c_{N_c}$ of training cases beforehand and uses them during the entire run (reported in [11], for example); program p's fitness is assessed using the formula $\frac{1}{N_c} \sum_{i=1}^{N_c} \mathrm{Err}(p, c_i)$. The individuals are therefore evaluated on just a fixed small subset of all possible input cases. This has advantages and disadvantages: all individuals are evaluated on the same inputs, so they can be directly compared in terms of performance; however, overfitting to a poorly chosen set of inputs could yield poorly performing individuals. We decided to select a small number of samples and to use the average for combining the results.

Sample Randomization involves modifying the training cases during the EC run. Again, program p's fitness is assessed using the formula $\frac{1}{N_c} \sum_{i=1}^{N_c} \mathrm{Err}(p, c_i)$, however the set $c_1, ..., c_{N_c}$ is randomized for each individual (*Random-Per-Individual* or *RPI*) or once per generation (*Random-Per-Generation* or *RPG*). The main advantage of the two methods is that new training cases are introduced all the time in the search process. However, because of the randomization, it may be difficult to compare and rank individuals: when performance is assessed on different training cases, a better individual may do worse just because its training case was more difficult. Random-Per-Generation was introduced in [12]; Random-Per-Individual was used, for example, when noise was added to improve robustness [10].

Coevolution (CVL) helped obtain very robust results for the particular domain of evolving sorting networks [3]; instead of having fixed or randomized training cases, Hillis experimented with coevolving them in a different population. Our implementation differs somewhat from the one presented in [3]: rather than having grid-worlds and locality notions, we decided for a much simpler implementation. More specifically, there are two populations: one with programs and one with sets of training cases. The fitness for program p is calculated as

$\frac{1}{N_c}\sum_{i=1}^{N_c}\text{Err}(p, C_i)$, where $C_1, ..., C_{N_c}$ represents the fittest set of training cases from the previous generation. Similarly, the fitness of the set of training cases $c_1, ..., c_{N_c}$ is calculated as $\frac{1}{N_c}\sum_{i=1}^{N_c}\text{Err}(P, c_i)$, where P is the best program from the previous generation. After assessing the fitness of programs and training cases, selection and breeding operators are used to create new populations of each kind. The coevolution of the two populations is simultaneous, and a random program and a random set of training cases are selected for evaluating the first generation.

Coevolution With Opponent Sharing (Coshare or CSH) is another coevolutionary method used when multiple populations are present. It has been first described in [13] under the name "competitive fitness sharing". The main idea is to treat every individual in the population as a resource and reward individuals that defeat opponents defeated by just few individuals. In this paper we consider two populations (one with programs and one with training cases) and we assume a complete mixing (each individual is tested on each training case). The fitness of a program p is calculated using the formula $\frac{1}{N_c}\sum_{i=1}^{N_c}\text{Err}(p, c_i)\frac{\text{Err}(p,c_i)}{\sum_{j=1}^{N_p}\text{Err}(p_j,c_i)}$, where N_c and N_p are the sizes of the populations of training cases, respectively of programs, and c_i and p_j represent individuals in the training case and program populations.

Fitness Sharing (FSH) implies that the weight of the performance of individuals on different training cases depends on the performance of the population on the specific training cases. The fitness of program p is calculated similarly as in the *Coshare* method by using the formula $\frac{1}{N_c}\sum_{i=1}^{N_c}\text{Err}(p, c_i)\frac{\text{Err}(p,c_i)}{\sum_{j=1}^{N_p}\text{Err}(p_j,c_i)}$, however $c_1, ..., c_{N_c}$ represents a set of training cases that is randomized every generation (as in the *Random-Per-Generation* method), rather than coevolved. A related approach is reported in [14].

4 Testing Robustness

One challenge to sampling is that the assessed quality of an individual is sensitive to its evaluation context (the set of input cases used). Thus to study the robustness of the programs, we used a variation of the *Train-Test-Validate* methodology [15,16]. In this methodology, individuals' fitnesses are assessed using a *training set*. The best-of-generation individuals are gathered, and the best-of-run individual is chosen from among them by testing each individual against a different *test set*. The final quality of the best-of-run individual is assessed using yet another different *validation set*. The size of these sets is relative to the frequency in which they are applied. Thus the validation set is larger than the test set, which in turn is larger than the training set. The use of the three separate sets means that the training, testing, and final comparison phases of the experiment are statistically independent. This is standard procedure in the Machine Learning and Data Mining communities, and as [17] argues, should be used more often in EC.

In the Artificial Ant, Multiplexer, and Even Parity problems, we also included results against an exhaustive sampling rather than a randomized validation set; our interest was in whether or not a large validation set was sufficient to justify not doing exhaustive sampling (particularly with infinite-sized input sets!). The validation sets, as it turns out, produced very similar results to the exhaustive samples.

We conducted the experiments on several common domains, all described below. More details on the domains and their standard settings can be found in [11]. All experiments were performed with the ECJ system [18]. Unless stated otherwise, the populations had 128 individuals and used 90% crossover probability, 10% reproduction probability, tournament selection with size 7, and one individual elitism. Other parameters were taken from [11]. Each experiment consisted of 50 independent runs. As we were testing generalization accuracy and not learning speed, we chose to give runs ample time (500 generations) to converge.

The problem domains used in this investigation are:

Symbolic Regression. The task is to learn the function $f(x) = x^4 + x^3 + x^2 + x$ from a set of pairs $\langle x, f(x)\rangle$ selected with x varying from -1 to 1. No ephemeral constants were used. The full input space is infinite in this problem. The training, testing and validation sets contained 20, 500, and 2000 random input cases respectively. The error on an input case was defined as the absolute distance between the real value of the generator function and the value estimated by the evolved program.

For the *Coevolution* method, the second population contained 128 training sets, each with 20 training cases; the training sets bred using Gaussian mutation ($\mu = 0, \sigma = 0.1$) with 100% probability, one-point crossover with 90% probability, and tournament selection of size 7. For the *Coshare* method, the second population had 20 individuals, each consisting of a single training case (a number between -1 and 1). The tournament selection had size 2, and Gaussian mutation ($\mu = 0, \sigma = 0.1$) was the only breeding operator.

11 Bit Multiplexer. The task is to search for boolean multiplexers that receive as input eight data and three address bits, and output the data bit corresponding to the specific address. The full input space has 2048 cases. The training, testing and validation sets contained 16, 32, and 64 random input cases respectively. The error on an input case was defined as 0 if the correct output was presented and 1 if not.

For the *Coevolution* method, the second population contained 128 training sets, each with 16 training cases. The training sets used bit-flip mutation (on average 2 bits were modified per training set) with 100% probability, one-point crossover with 90% probability, and tournament selection of size 7. For the *Coshare* method, the second population had 16 individuals, each consisting of a single training case. *Coshare* used tournament selection of size 2, followed by bit-flip mutation (on average, 3 bits per training case were modified).

Artificial Ant. The standard GP Artificial Ant problem consists of a two-dimensional discretized toroidal environment which contains a broken trail of

Table 1. Statistical significance groupings for the best-of-run performances for all problem domains. Horizontal bars at the same level indicate techniques with statistically insignificant differences in means. Methods are ordered according to the performance of the best-of-run individuals from best (leftmost) to worst (rightmost) for each domain. The Validation Ranking is obtained by evaluating the best-of-run individuals on randomly generated validation sets. The Exhaustive Ranking is obtained by evaluating the best-of-run individuals on all possible input cases

	Validation Ranking						Exhaustive Ranking					
Regression	CVL	RPG	FSH	FRI	RPI	CSH						
Multiplexer	RPI	FSH	FRI	RPG	CSH	CVL	RPI	FSH	FRI	CSH	RPG	CVL
Ant	CVL	CSH	RPG	FSH	RPI	FRI	CVL	CSH	RPG	FSH	RPI	FRI
4 Bit Parity	CSH	FSH	RPG	RPI	FRI	CVL	CSH	FSH	RPG	RPI	FRI	CVL

food pellets (we used the "Santa Fe" trail map). The GP program controls an ant starting at an initial position and orientation, which tries to collect as much food as possible in a fixed number of steps. Our experiment deviates from the standard by allowing the ant to start from any position and have any initial orientation. The full input space has 4096 possible input cases (triplets $\langle x, y, o \rangle$ where x and y range between 0 and 31, and o may take any of the four orientations). The training, testing and validation sets contained 10, 100 and 500 input cases each. The error on an input case was defined as the amount of food not consumed.

For the *Coevolution* method, the second population contained 128 training sets, each with 10 training cases; this population used a 3.4% mutation (about one gene per training set was randomized), 90% one-point crossover, and tournament selection of size 7. For the *Coshare* method, the second population had 10 individuals, each consisting of an training case (3 genes). *Coshare* again used tournament selection with size 2, this time followed by mutation with 34% probability of randomizing a gene, plus one-point crossover with 90% probability.

4-Bit Even Parity. The task consists of learning the parity of a bit string without counting the bits, using only the four boolean functions AND, OR, NAND, and NOR. 4-bit Even Parity has a small input space (16 cases). However the task is interesting in that it is difficult to generalize from a reduced sample (in our case, 8 cases in the training and 16 in the testing sets). The small input size also gave us a chance to see what would happen if the training, testing, and validation sets did not uniformly sample the space: in particular, the validation set was done on 24 random input cases. This is larger than the input space and by design cannot uniformly sample it.

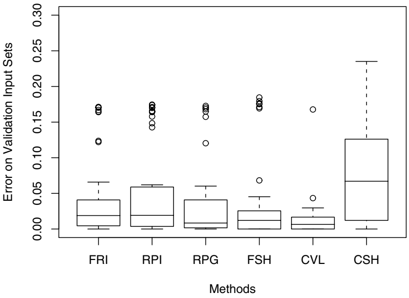

Fig. 1. Symbolic Regression: Boxplot of errors of methods on validation sets.

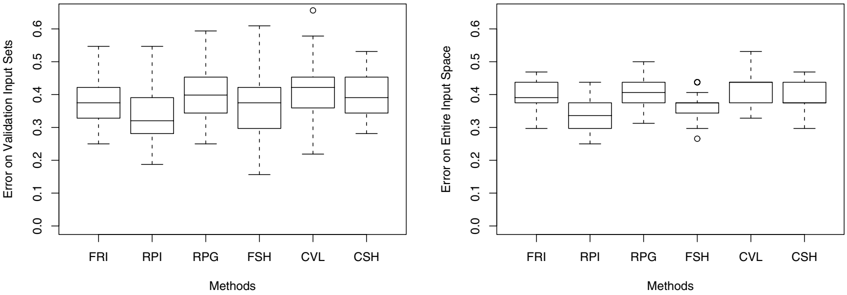

Fig. 2. 11-Bit Multiplexer: Boxplots of errors of methods on validation sets (left) and on all possible input cases (right).

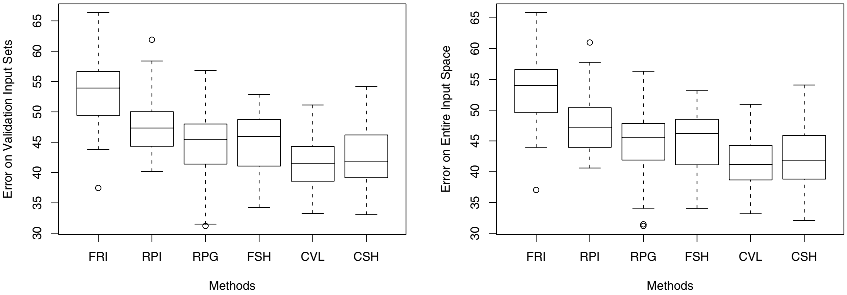

Fig. 3. Artificial Ant: Boxplots of errors of methods on validation sets (left) and on all possible input cases (right).

The *Coevolution* method contained 128 tests sets in the the second population, each with 8 training cases, using tournament selection of size 7, and bit-flip mutation that changed on average three bits per training set and one-point crossover with 90% probability. In the *Coshare* method, the second population had 8 individuals, each consisting of an training case. *Coshare* used tournament selection of size 2, followed by mutation flipping 2 bits per training case on average.

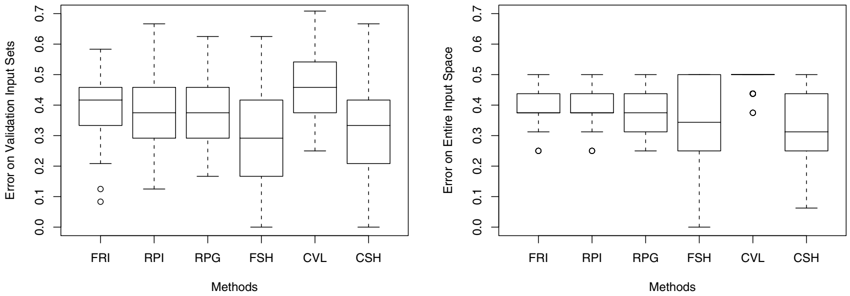

Fig. 4. 4-Bit Even Parity: Boxplots of errors of methods on validation sets (left) and on all possible input cases (right).

5 Results

Our results compare the best-of-run individuals on the validation sets. We initially used one-way ANOVA tests at 95% confidence, followed by Tukey post-hoc tests for ranking the methods. A more careful analysis of the data indicated that the results are normally distributed in the Artificial Ant, 11-Bit Multiplexer and 4-Parity domains, but they are not normally distributed in the Symbolic Regression domain. Additionally, the results have different variances, which may pose problems for the Tukey tests which assume equal variances. We compared the methods by pairwise comparisons using Welch's two-sample tests combined with Boole's inequality[1]. For 95% confidence when comparing 6 methods, 15 two-sample pairwise tests need to be performed at confidence $1 - \frac{1-0.95}{15} = 99.67\%$ each. We confirmed the ranking in the Symbolic Regression domain by performing comparisons using a nonparametric test due to Steel and Dwass.

Figures 1–4 present boxplots of the distributions of error among the best-of-run individuals on the validation set of each method. Except for Symbolic Regression, which has an infinite input space, the distributions on the exhaustive case spaces are also given. The center line inside each box plot represents the median (not the mean) of the sample.

As shown in Table 1, no one method is superior to the others over all domains in the study. In the Symbolic Regression domain, *Coevolution*, *Random-Per-Generation*, and *Fitness-Sharing* performed best. *Coevolution* and *Coshare* method did best in the Artificial Ant problem[2]. In the boolean function domains, the results are different. *Random-Per-Individual* and *Fitness-Sharing* perform

[1] Welch's two-sample test is similar to the standard t-test, but the possibly unequal variances of the two distributions are separately approximated. Boole's inequality is used for repeated testing of the same data sets, and it mainly increases the confidence requirements for each individual test proportional to the number of tests performed.

[2] In the Ant domain, *Random-Per-Individual* has significantly worse performance than *Fitness-Sharing*, but there is not enough confidence to state it is also worse than *Random-Per-Generation* which has a large variance in results.

best in the 11-Bit Multiplexer domain, and *Coshare* and *Fitness-Sharing* perform best in the 4-Bit Even Parity.

We analyzed the average performance of the best-of-generation and best-of-run individuals on the training, test and validation sets (graphs not shown). In general, *Coevolution* and *Coshare* both have higher errors on the training sets than on the validation sets; this is indicative of coevolution's relative success in discovering and promoting difficult training sets. We also observed overfitting to the testing sets with *Random-Per-Generation*, *Random-Per-Individual* and *Fixed-Random-Initial*. By comparing the validation and test results with a t-test at 95%, we found that in the Artificial Ant domain, these three methods all performed significantly worse on the validation sets than on the test sets.

We found that the Train-Test-Validate methodology as used in this paper gives a good approximation of the robustness of the evolved programs over the entire input space. As shown in Table 1, the ordering is relatively similar between validation and exhaustive rankings.

We finish with some speculation as to why various methods performed the way they did.

The four domains can be divided into two categories: in the boolean problem domains, it is easy to discover a difficult input case; whereas in the Artificial Ant and Regression problems, it is relatively more challenging to discover one. This is because Regression does not usually have large jumps with slightly modified X values, and small changes in orientation or location do not usually affect Artificial Ant solutions radically. We believe this may explain the behavior of *Coevolution*, which did poorly in the boolean problems but well in the other two. Specifically, we suspect that *Coevolution*'s population of training cases adapted too rapidly to difficult problems for the GP population to solve, thus contributing to a loss of gradient.

To quickly test this hypothesis, we changed the population of training cases in the 4-bit Parity domain to contain just two individuals (effectively hillclimbing), with size 2 tournament selection, and a mutation rate smaller by one order of magnitude. The goal was to prevent the training cases from improving too rapidly. The resulting solutions had statistically significantly better performance on both the validation and exhaustive sets of cases.

The Symbolic Regression domain differs from other problems in that each input case can cause an arbitrarily large error. This particularly affects the *Coshare* and *Fitness-Sharing* methods, in which a moderately large, but unique error in a single test case can ruin the fitness of an individual even if it significantly outperforms mediocre peers in the population on all the other cases.

The *Fixed-Random-Initial* method never fell into the top tier, suggesting that it should not be used as the first choice method when robust results are desired.

6 Conclusions and Future Work

Methods for sampling large input spaces may have particular utility to the evolution of computer algorithms and functions. In this initial investigation of the issue, we compared several approaches to sampling this space in the context of four genetic programming domains. However, we observed that *which* approaches did best was dependent on problem domain features. For example, *Coevolution* performed well in the Symbolic Regression and Artificial Ant domains, but it had the worst results in the 11-Bit Multiplexer and 4-Bit Even Parity. *Fitness-Sharing* fell in the first tier in the Symbolic Regression and 4-Bit Even Parity, ant fairer reasonably well (second tier) in the 11-Bit Multiplexer and the Artificial Ant problems.

Part of this may be due to unforeseen consequences of how fitness is assessed (such as Symbolic Regression). This brings up an important question to be addressed in future work: when the performance of an individual is assessed over several input cases, how can its overall performance be approximated? Possible alternatives to averaging may use the mode, median, maximum or minimum values, or include standard deviation or interquartile information ([10] in particular suggests using the minimum of several noisy evaluations). Additional research is also required to identify when overfitting occurs and how it can be avoided.

Acknowledgements. This research was partially supported through a gift from SRA International and through Department of Army grant DAAB07-01-9-L504. The authors would like to thank Dr. Daniel Menasce, R. Paul Wiegand, Elena Popovici, Gabriel Balan and Marcel Barbulescu for their help in conducting the research investigation. We would also like to thank Dr. Clifton Sutton for his assistance in analyzing the data.

References

1. Bersano-Begey, T.F., Daida, J.M.: A discussion on generality and robustness and a framework for fitness set construction in Genetic Programming to promote robustness. In Koza, J.R., ed.: Late Breaking Papers at the 1997 Genetic Programming Conference, Stanford University, CA, USA, Stanford Bookstore (1997) 11–18
2. Pagie, L., Hogeweg, P.: Evolutionary consequences of coevolving targets. Evolutionary Computation **5** (1997) 401–418
3. Hillis, D.: Co-evolving parasites improve simulated evolution as an optimization procedure. Artificial Life II, SFI Studies in the Sciences of Complexity **10** (1991) 313–324
4. Kushchu, I.: Genetic Programming and evolutionary generalization. IEEE Transactions on Evolutionary Computation **6** (2002) 431–442
5. Juille, H., Pollack, J.: Coevolutionary arms race improves generalization. In Koza, J.R., ed.: Late Breaking Papers at the Genetic Programming 1998 Conference, University of Wisconsin, Madison, Wisconsin, USA, Stanford University Bookstore (1998)
6. Kinnear, Jr., K.E.: Generality and difficulty in Genetic Programming: evolving a sort. In Forrest, S., ed.: Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, University of Illinois at Urbana-Champaign, Morgan Kaufmann (1993) 287–294

7. Cavaretta, M.J., Chellapilla, K.: Data mining using Genetic Programming: The implications of parsimony on generalization error. In Angeline, P.J., Michalewicz, Z., Schoenauer, M., Yao, X., Zalzala, A., eds.: Proceedings of the Congress on Evolutionary Computation. Volume 2., Mayflower Hotel, Washington D.C., USA, IEEE Press (1999) 1330–1337
8. Rosca, J.: Generality versus size in Genetic Programming. In Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (1996) 381–387
9. Droste, S.: Efficient Genetic Programming for finding good generalizing boolean functions. In Koza, J.R., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M., Iba, H., Riolo, R.L., eds.: Genetic Programming 1997: Proceedings of the Second Annual Conference, Stanford University, CA, USA, Morgan Kaufmann (1997) 82–87
10. Reynolds, C.W.: Evolution of corridor following behavior in a noisy world. In: Simulation of Adaptive Behaviour (SAB-94). (1994)
11. Koza, J.: Genetic Programming: on the programming of computers by means of natural selection. MIT Press (1992)
12. Moore, F.W., Garcia, O.N.: New methodology for reducing brittleness in Genetic Programming. In Pohl, E., ed.: Proceedings of the National Aerospace and Electronics 1997 Conference (NAECON-97), IEEE Press (1997)
13. Rosin, C., Belew, R.: New methods for competitive coevolution. Evolutionary Computation **5** (1997) 1–29
14. Forrest, S., Smith, R.E., Javornik, B., Perelson, A.S.: Using Genetic Algorithms to explore pattern recognition in the immune system. Evolutionary Computation **1** (1993) 191–211
15. Rowland, J.: On model selection in supervised learning: Do we really know when to stop? In: Evolutionary and Neural Computation in Bioinformatics: A PPSN VII Workshop. (2002)
16. Brameier, M., Banzhaf, W.: A comparison of Linear Genetic Programming and Neural Networks in medical data mining. IEEE Transactions on Evolutionary Computation **5** (2001) 17–26
17. Eiben, A.E., Jelasity, M.: A critical note on experimental research methodology in EC. In: Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002). (2002) 582–587
18. Luke, S. ECJ 9: An Evolutionary Computation research system in Java. Available at http://www.cs.umd.edu/projects/plus/ec/ecj/ (2002)