Alternative Evolutionary Algorithms for Evolving Programs: Evolution Strategies and Steady State GP

Darrell Whitley, Marc Richards
and Ross Beveridge
Computer Science, Colorado State University
Fort Collins, CO 80524
whitley or ross @cs.colostate.edu

Andre' da Motta Salles Barreto
Programa de Engenharia Civil/COPPE,
Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brazil
andremsb@coc.ufrj.br

ABSTRACT

In contrast with the diverse array of genetic algorithms, the Genetic Programming (GP) paradigm is usually applied in a relatively uniform manner. Heuristics have developed over time as to which replacement strategies and selection methods are best. The question addressed in this paper is relatively simple: since there are so many variants of evolutionary algorithm, how well do some of the other well known forms of evolutionary algorithm perform when used to evolve programs trees using s-expressions as the representation? Our results suggest a wide range of evolutionary algorithms are all equally good at evolving programs, including the simplest evolution strategies.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: [Program Synthesis]

General Terms

Experimentation, Performance

Keywords

Genetic Programming, Steady-State Genetic Algorithms, Evolution Strategies

1. INTRODUCTION

Over the last 10 to 15 years, genetic programming has become one of the major subdisciplines within the genetic and evolutionary-computation community. In genetic programming (GP) the individuals that undergo evolution are typically hierarchically-structured computer programs [6]. An intuitive way to represent this kind of data structure is to use a parse tree whose terminal leafs are the operands and the internal ones are the operators (see Figure 1).

From both an historical and a practical point of view, genetic programming borrows many fundamental mechanisms from traditional genetic algorithms. By traditional genetic algorithms, we refer to those originally described by Holland [5] as well as Goldberg's "Simple Genetic Algorithm"[3]. One of the characteristics shared by genetic programming and traditional genetic algorithms is the *generational* replacement scheme: the new generation of offspring replaces the previous generation of parents. This is not strictly necessary, though. Steady state genetic algorithms do not use generational replacement, but keep the best solutions found so far.[1] For parameter optimization and combinatorial optimization problems, *steady-state* genetic algorithms appear to yield better solutions than generational approaches [16]. Richards et al. [12] used the Genitor steady-state genetic algorithm to evolve program trees for an Unmanned Aerial Vehicle (UAV) control application; the evolved program trees control the behavior of a team of UAVs flying survey missions over a target area with hidden hazards and the potential for losing aircraft. Richards et al. report that the "Genitor" steady state implementation of GP produced better results than a traditional generational GP system.

The results of Richards et al. suggest a simple but natural question: how well do some of the other well known forms of evolutionary algorithm perform when used to evolve programs trees using Lisp-like s-expressions as the representation? This question then invites related questions concerning the use of mutation versus recombination in this search

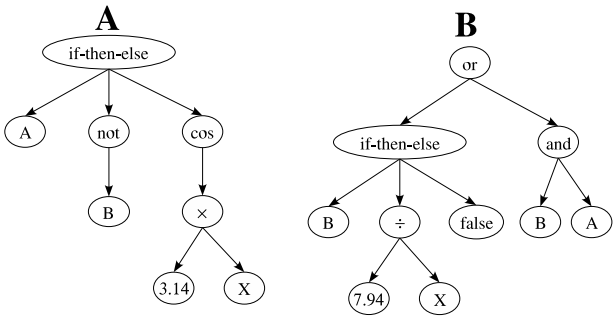

Figure 1: GP individuals represented as parse trees

[1]The name "steady state" introduced by Gil Syswerda [13] is somewhat unfortunate, since Syswerda found "steady state GAs" do not display steady state behavior; Alden Wright has suggested a more accurate name is "monotonic genetic algorithms."

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

space, as well as questions about selective pressure. Going from generational GP to steady state GP is a relatively minor code change: the recombination and mutation operators used do not change.

One of the main contributions of this paper is the use of relatively standard mutation based evolution strategies using a wide range of selection and population choices. Besides the replacement strategy, parameters such as population size and the selection pressure will have an impact on the performance of the evolutionary algorithms. An exhaustive search over the parameter space is not feasible, so we are left to wonder if perhaps another combination of parameters would result in better performance. We really cannot resolve this problem in the current paper, but we do test different population sizes and vary the selective pressure.

Some of the questions associated with the use of evolution strategies have been addressed before, but in a different, piecemeal fashion. We review key background experiments in the next section. Our results sometime conform with previous work and reinforce common practices. Other results were somewhat surprising: the simple (1,10)ES produced the best overall results.

2. BACKGROUND

Genetic programming almost always refers to what might be called "Koza-Style" genetic programming. Perhaps this is because Koza's work presents such a good, easy to use "template" enabling one to quickly implement a proven approach to GP [6].

In the current paper we will ignore alternative representations, perhaps most notable of which is the linear genetic programming representations used by systems such as AIM-GP: Automatic Induction of Machine-code Genetic Programming. Instead of operating on trees, AIM-GP uses C code operations to act directly on registers. This means that, in effect, AIM-GP generates a subset of C as its program output [1].

Our first question is this: *Can steady state genetic programming work as well or better than the generational GP model?*

This is a question that Koza explored in 1994 [6] and other researchers have occasionally used a steady-state form of genetic programming. For example, all the GP experiments in Langdon's book [8] *Genetic Programming and Data Structures* used steady-state replacement.

This question highlights another issue however. Genetic Programming typically uses tournament selection with the tournament size set at 7. Tournament size will be denoted by **T** in the paper. This means when parents are being selected for recombination, 7 individuals are drawn from the population, and the best of these is allowed to reproduce; another tournament of T=7 individuals is used to determine the second parent. In regular genetic algorithms the tournament size is often T=2, producing far less selective pressure. In steady-state genetic algorithms the selective pressure is also usually 2 or less. However, Goldberg and Deb [4] showed that because steady-state genetic algorithms use a selective strategy similar to truncation selection, when coupled with regular selection the actual selective pressure is much greater than the tournament size or the nominal selective pressure would suggest. In fact, in terms of the time it takes for the best individual to take over the population under selection, T=2 under the steady state model behaves more like T=7 under the generational model. Selective pressure less than 2 is often used with steady state genetic algorithms.

Our second question is: *How well do evolution strategies evolve program trees?*

This question is more complicated, because it suggests a potential change in focus with regard to operators: can a "mutation-only" search of the program tree space be as effective as one based on recombination? In fact several researchers have addressed this issue. Of those which are more noteworthy, in 1994 O'Reilly and Oppacher presented results that indicated a stochastic hill-climber using only mutation could be competitive with traditional GP [11]. In 1997 Chellapilla [2] reported that a population based form of evolutionary programming using only mutation produced results similar to traditional generational genetic programming. A study by Luke and Spector in 1998 [10] suggests there may not be a simple answer concerning the merits of crossover versus mutation. There was some evidence that mutation may work better in small populations and crossover in large populations.

There are two basic forms of evolution strategies (ES). In the $(\mu + \lambda)$-ES the μ best of the combined parent and offspring generations are retained using truncation selection, somewhat like a steady-state genetic algorithm. In the (μ, λ)-ES the μ best of the λ offspring replace the parents, much like a generational genetic algorithm. The definitions of the $(\mu + \lambda)$-ES and the (μ, λ)-ES predates the distinction between generational and steady state genetic algorithms.

Another interesting question is whether we really need a population at all. One basic form of evolution strategy only keeps the best solution seen so far; in fact, the $(1 + \lambda)$ES can be seen as a type of greedy stochastic local search. It is stochastic because there is no fixed neighborhood and therefore the neighborhood does not define a fixed set of local optima, but otherwise it is like local search: sample the neighborhood and move to the best point. Can a search that does not use a population (i.e., $\mu = 1$) successfully be used to discover program trees? This echos O'Reilly and Oppacher's early experiments with stochastic hill-climbing. And what implications does this have with regard to problems such as code bloat?

Finally, in evolution strategies it is common for the number of offspring produced (λ) to be larger than the parent population (μ). What impact does this have? The question has received some attention in the Genetic Programming community. Tackett [14] explored the use of *brood selection*, specifically implemented as Greedy Recombination. In its simplest form, the brood is of size 2N where N is the population size and crossover produces 2 offspring and both are initially kept. Tackett also notes that Genetic Programming could be posed as a state space tree search, and that population based methods could be reexpressed as a form of beam search. The main conclusions of Tackett's work is that the use of brood selection could allow the use of smaller populations. In general, Tackett's work produced more questions than answers, but these still remain very good questions.

The current paper is not meant to provide definitive answers. Rather, it re-opens these questions from a slightly different perspective: what happens when we apply standard, well-established evolutionary algorithms to the problem of evolving program trees? The empirical results may challenge some preconceptions and are an invitation for debate and further research.

3. CONFIGURATION OF EXPERIMENTS

The next subsections discuss the test problems and various implementation details.

3.1 Test Problems

We use three relatively standard test problems: the artificial ant problem, the 11-multiplexer and a symbolic regression problem. After some preliminary analysis of trends, we added a fourth problem: the pole balancing and cart centering problem. The following paragraphs describe the problems used in the experiments as well as the fixed parameters.

3.1.1 Artificial Ant

The artificial ant problem is formulated as follows: There is a simulated agent (the "ant") operating over a toroidal 2D grid. The grid contains a trail with "food" objects placed at irregular intervals. The ant is capable of moving forward, turning 90 degrees left or right, and sensing food that is directly in front of it. The ant "eats" a piece of food by moving onto the grid square containing it, at which point the food is removed from the square. The goal is for the ant to eat all the food on the trail while taking the fewest number of actions. To apply GP to this problem, a computer program that will control the ant is evolved. A typical artificial ant scenario will use a 32 x 32 unit grid and allow up to 600 actions before terminating. The fitness score is the total amount of food consumed by the ant (i.e., minimize the amount of unconsumed food). For an analysis of the artificial ant problem, the reader is referred to [9].

3.1.2 11-Multiplexer

The 11-Multiplexer problem involves 3 address bits that map to the contents of 8 content bits. The terminals set is made up of the values of the address bits, a0, a1, a2 and of the variables d0 to d7. The problem is to find a Boolean function that executes multiplexing over the 3-bit address. Given a string of bits, the function should return the correct variable which corresponds to the setting of the address bits. Since there are 11 bits altogether, there are 2048 test cases.

3.1.3 Symbolic Regression

In symbolic regression, a mathematical formula is evolved to fit a polynomial expression to a set of known Cartesian points. Typical GP functions are x, +, -, and *, and the fitness score is usually computed as the mean squared error of the evolved expression evaluated over a fixed range. The target symbolic regression example used in this paper was $x^6 - 2x^4 + x^2$ over the range -1 to 1, as used in Koza II [7]. Symbolic regression is defined over a continuous space, whereas problems such as the 11-multiplexer is defined over a discrete finite space.

3.1.4 Pole balancing and Cart Centering

In the pole balancing and cart centering task the goal is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over. The track is of finite length, so the controller must also avoid hitting the end of the track. The fitness of an individual is based upon how many times, out of 50 randomized trials, the individual is able to keep stay within the limits of the track and keep the pole balanced for $1,000$ time steps. Each trial begins with the cart's configuration chosen at random.

The terminal set in this problem consists of the position and velocity of the cart along the track, the angle between the cart and the pole and the angular velocity of the pole, as in [15]. This information is combined in the Genetic Programming trees via standard arithmetic operators, and depending on the sign of the output value a push to the right or to the left is performed.

3.2 Algorithms and Parameter Settings

3.2.1 Traditional Genetic Programming

Our traditional generational genetic programing code is based on Sean Luke's ECJ software environment. We use 4 different population sizes: 250, 500, 1000 and 2000. We use tournament selection to implement selection. A good deal of literature suggests that genetic programming works best with a tournament size of T=7. Genetic algorithms for optimization commonly use lower selective pressure: typically T=2. Both T=2 and T=7 were tested in our experiments.

All of the terminals and functions used in our experiments are the standard ones used with the standard benchmark problems. While pole balancing and cart centering is not as common as a benchmark, it is a problem that Koza has used genetic programming to solve.

No mutation was used in the experiments reported here. We experimented with various types and levels of mutation, but saw no improvement in the results.

3.2.2 Steady State Genetic Programming

Implementing the steady state form of genetic programming was simple; we used the Genitor model. We again used population sizes of 250, 500, 1000 and 2000. After the population is first evaluated, it is sorted. Tournament selection is used to select two parents for recombination. We used both T=2 and T=7 to mirror the traditional GP experiments (even though these will generate greater selective pressure under the steady state model). Two parents generate one offspring; this offspring replaces the worst member of the population. The new offspring is bubbled into place so that the population remains sorted. The process is then repeated. The recombination operator remains the same.

3.2.3 Evolution strategies

Because evolution strategies are fundamentally different from genetic algorithms in many ways, the evolution strategies implemented here also differ from the generational and steady state genetic programming approaches. We use four combinations of μ and λ; paralleling the generational and steady-state approaches, we implemented both comma and plus strategies: the resulting eight ES combinations were as follows:

(1,10)-ES (50,250)-ES (100,500)-ES (200,1000)-ES
(1+10)-ES (50+250)-ES (100+500)-ES (200+1000)-ES

We experimented with two forms of mutation: node mutation and subtree mutation. Node mutation changes a single point in the tree: either an internal function is changed or a terminal leaf node is changed. Node mutation is obviously conservative. Subtree mutation is meant to be more like recombination: a subtree is selected as if it were being selected for recombination, except that instead a new subtree is randomly generated (using "grow") to replace the old subtree. We found subtree mutation to be more effective, and thus used it in all of our experiments.

4. EXPERIMENTS AND RESULTS

All experiments were run for 100,000 evaluations. We used 20 runs for each experiment for each problem. Our preliminary experiments were run on the ant problem, the 11-multiplexer and the symbolic regression problem.

We had certain hypotheses (or at least expectations) before conducting these experiments. In the box-whisker plots, major variants are marked A and B, where our expectation initially was that A would be better than B. Thus, for the steady-state GP, the smaller tournament T=2 (ss A) was expected to be better than T=7 (ss B). For the generational GP, the reverse was expected, with T=7 (gen A) expected to be better than T=2 (gen B). For the evolutionary strategies, we expected the "plus" evolutionary strategies (es A) to be better than the comma-ES (es B).

The results are shown in figure 2. A "box and whiskers" plot summary of the data is presented. The black dot is the position of the median. The height of the solid white box is the inter quartile range (IQR) which includes the central 50% of the data. The whiskers extend below and above the white box to a distance of 1.5 times the IQR. However, they must terminate on a data point, so the whiskers may be less than 1.5 times the IQR. This means in some cases the whiskers disappear in our data.

The population sizes in the box-whisker plots are denoted T, S, M, L corresponding to tiny, small, medium and large. For generational and steady state GP, these correspond to 250, 500, 1000 and 2000. For the evolution strategies these correspond to the population sizes λ and associated μ values given in section 3.2.3.

We do not explicitly test for statistical significance. Given our 16 versions of algorithms and 4 problems, one would expect 3 of the 64 results (i.e., 4.6 percent) to be "significant" even if the results were purely random. However, the whisker plots suggest when differences are most likely significant and when the results appear to be similar.

The first important observation is that all of the evolutionary algorithms do relatively well. Overall, the generational genetic programming paradigm using a tournament size of T=2 stands out as worst. This supports the common notion that GP works best with a tournament size of T=7.

As expected, for steady state GP, a tournament size of T=2 is best, although the tournament size does not have as much of an impact on steady state GP as it does on generational GP. The steady state genetic programming results are arguably just as good as the generational GP with T=7.

Perhaps the most important outcome is how well the various evolution strategies performed relative to both steady state and generational genetic programming. It is further surprising to see that the "comma" evolution strategies were often better than the "plus" evolution strategies. Even more surprising, the tiny (1,10)-ES yielded the best overall performance across all algorithms. This is surprising because in effect there is no population and the search is a form of stochastic local search.

While it was somewhat surprising that the (μ, λ)-ES was actually somewhat better than the $(\mu+\lambda)$-ES, this does echo the fact that the generational GP approach typically yields very good performance. There seems to be some advantage associated with the mobility that comes from turning over and replacing the population, or in the case of the (1,10)-ES, of moving to the best point in the stochastic neighborhood even if this is not an improving move.

	ant	multi11	symb	pole
gen GP t2 250	28.30	238.35	0.228	221.20
gen GP t2 500	30.35	228.60	0.124	229.05
gen GP t2 1000	27.00	277.35	0.202	251.95
gen GP t7 250	29.55	122.00	0.122	388.15
gen GP t7 500	22.05	107.20	0.117	312.50
gen t7 1000	**16.30**	**72.00**	**0.058**	**249.25**
ss GP t2 250	28.20	177.40	0.104	370.25
ss GP t2 500	25.90	101.60	0.055	292.15
ss t2 1000	**15.25**	**111.70**	**0.049**	**214.85**
ss GP t7 250	30.05	186.05	0.171	491.10
ss GP t7 500	27.30	195.20	0.119	406.95
ss GP t7 1000	16.05	153.60	0.074	291.25
(1,10)-ES	**14.70**	**63.60**	**0.039**	**193.05**
(50,250)-ES	25.30	93.60	0.049	283.20
(100,500)-ES	23.35	57.85	0.058	265.25
(200,1000)-ES	23.35	85.00	0.052	256.40
(1+10)-ES	21.55	69.60	0.156	276.40
(50+250)-ES	20.30	110.00	0.054	214.35
(100+500)-ES	15.70	114.80	0.061	237.75
(200+1000)-ES	12.00	79.20	0.034	227.60

Table 1: Final results obtained by select configurations of the GP algorithm. For the ant, multiplexer and symbolic regression problem, the reported number is the residual error. The reported results for the pole-balancing task correspond to how many out of 625 random initial states the best individual of the run failed to balance the pole for at least 1,000 time steps. All the results were averaged over 20 runs.

Loss of genetic diversity is a common problem with traditional genetic algorithms. It can also be a problem with traditional genetic programming and is often even worst when steady state approaches are used. Loss of diversity would seem to be less of an issue with the evolution strategies, since the mutation operator continues to explore the search space. Of course, evolving program trees is very different from evolving bit strings, or real valued vectors or permutations, as is common in parameter and combinatorial optimization problems.

4.1 Adding another test domain

Based on these results, we expanded our experiments to include the pole balancing and cart centering problem. We wanted to see if another problem showed the same kinds of trends.

Table 1 presents the average best solution found by select algorithms on each problem. To focus the reader on the most important trends, we display in bold face the generational GP with population size 1000 and T=7 as well as the steady state GP with population size 1000 and T=2. We also bold face the (1,10)ES, which appears to have produced the best overall performance. All problems are minimization problems. The differences are often not statistically significant in the current study. The 2000 population generational and steady state GP results are not shown because these were no better than the 1000 population results.

The pole balancing problem showed several trends that are different from the other problems. Generational GP with T=7 and a population of 1000 was not particularly good for

Algorithm	Ant	Multi11
gen GP, T=7, 1000	1	8
ss GP, T=2, 1000	4	4
$(1, 10)$-ES	1	11
$(1 + 10)$-ES	3	8

Table 2: These results report how many time a problem was solved by a particular method.

the pole balancing problem; generational GP using a population size of either 500 or 250 with T=2 was better. The steady state GP with T=2 and population size 1000 was also better than generational GP with T=7 and a population of 1000. This leads us to conjecture that the results reported by Richard's et al. could have been a function of the application domain, as well as a function of the population and tournament size.

Overall, the (1,10)-ES still produced the best results.

We were also curious how often a particular method exactly "solved" a particular problem. None exactly solved the symbolic regression problem, and the pole balancing problem does not have a well-defined exact solution. Thus table 2 reports how often the ant and 11-multiplexer problems were solved by the best algorithms in each class.

5. DISCUSSION AND CONCLUSIONS

Genetic Programming is usually applied in a relatively uniform manner, with decisions like replacement strategy and selection method based on rules of thumb established over the years. This has been fundamental in the establishment of GP's methodology and terminology; conventions like Koza's "five steps to use genetic programming" [6] standardizes and facilitates the application of GP to new problems.

Our results show few differences between different evolutionary algorithms. In other ways our results are similar to those of Luke and Spector in as much as mutation-only evolution strategies are often able to work with smaller population sizes. Unlike the study by Luke and Spector, we did not find that crossover was somewhat better than mutation for symbolic regression.

The most important finding is the relatively strong performance demonstrated by the (1,10)-ES. This suggests that simple local search methods can also be used to evolve programs. Usually, "code bloat" is a major factor limiting the use of genetic programming. An extremely interesting question is whether the use of a (1,10)-ES would necessarily display the same "code bloat" problem associated with traditional GP.

There are also many other questions suggested by these results: Do the evolution strategies scale up to work in more complex domains? What kinds of new and different forms of parallel and portfolio methods are possible given the use of these different forms of evolutionary algorithms? What kinds of new hybrid methods might be developed by combining approaches?

These are all interesting and worthwhile avenues for future research.

6. REFERENCES

[1] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming: An Introduction*. Morgan Kaufmann, San Francisco, CA, 1998.

[2] K. Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, 1997.

[3] D. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Reading, MA, 1989.

[4] D. Goldberg and K. Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms 1*, pages 69–93. Morgan Kaufmann, 1991.

[5] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

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

[7] J. R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT Press, Cambridge, MA, USA, 1994.

[8] W. B. Langdon. *Genetic Programming and Data Structures*. Kluwer Academic Publishers, 1998.

[9] W. B. Langdon and R. Poli. Why ants are hard. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Morgan Kaufmann, July 1998.

[10] S. Luke and L. Spector. A revised comparison of crossover and mutation in genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Morgan Kaufmann, July 1998.

[11] U. O'Reilly and F. Oppacher. Program search with a hierarchical variable length representation. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature – PPSN III*, pages 397–406, Berlin, 1994. Springer.

[12] M. Richards, D. Whitley, R. Beveridge, T. Mytkowicz, D. Nguyen, and D. Rome. Evolving cooperative strategies for uav teams. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2006)*. ACM Press, 2006.

[13] G. Syswerda. Reproduction in Generational and Steady State Genetic Algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms 1*, pages 94–101. Morgan Kaufmann, 1991.

[14] W. Tackett. Greedy Recombination and Genetic Search on the Space of computer Programs. In D. Whitley and M. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 271–297. Morgan Kaufmann, 1995.

[15] D. Whitley, S. Dominic, R. Das, and C. W. Anderson. Genetic reinforcement learning for neurocontrol problems. *Mach. Learn.*, 13(2-3):259–284, 1993.

[16] L. D. Whitley, T. Starkweather, and D. Shaner. The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 22, pages 350–372. Van Nostrand Reinhold, New York, 1991.

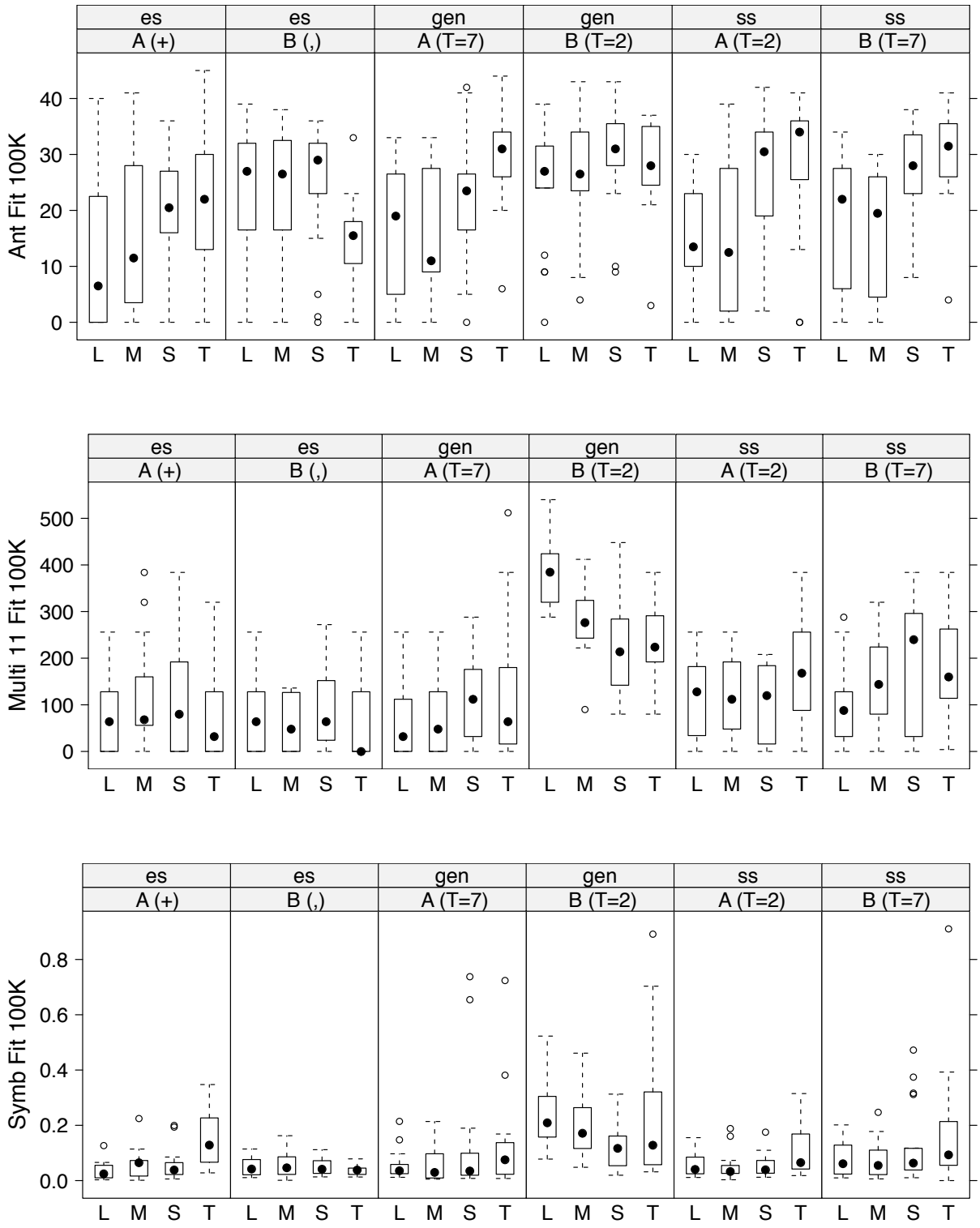

Figure 2: Whisker plots summarizing results for the ant, multiplexer and symbolic regression tasks. The vertical axis represents fitness in all cases. The three types of algorithms are: evolution strategies (es), generational genetic programming (gen) and steady-state genetic programming (ss). Four population sizes are shown: large (L), medium (M), small (S) and tiny (T). Two variants for each algorithm, A and B, are shown. These variants and population sizes are fully explained in the running text.

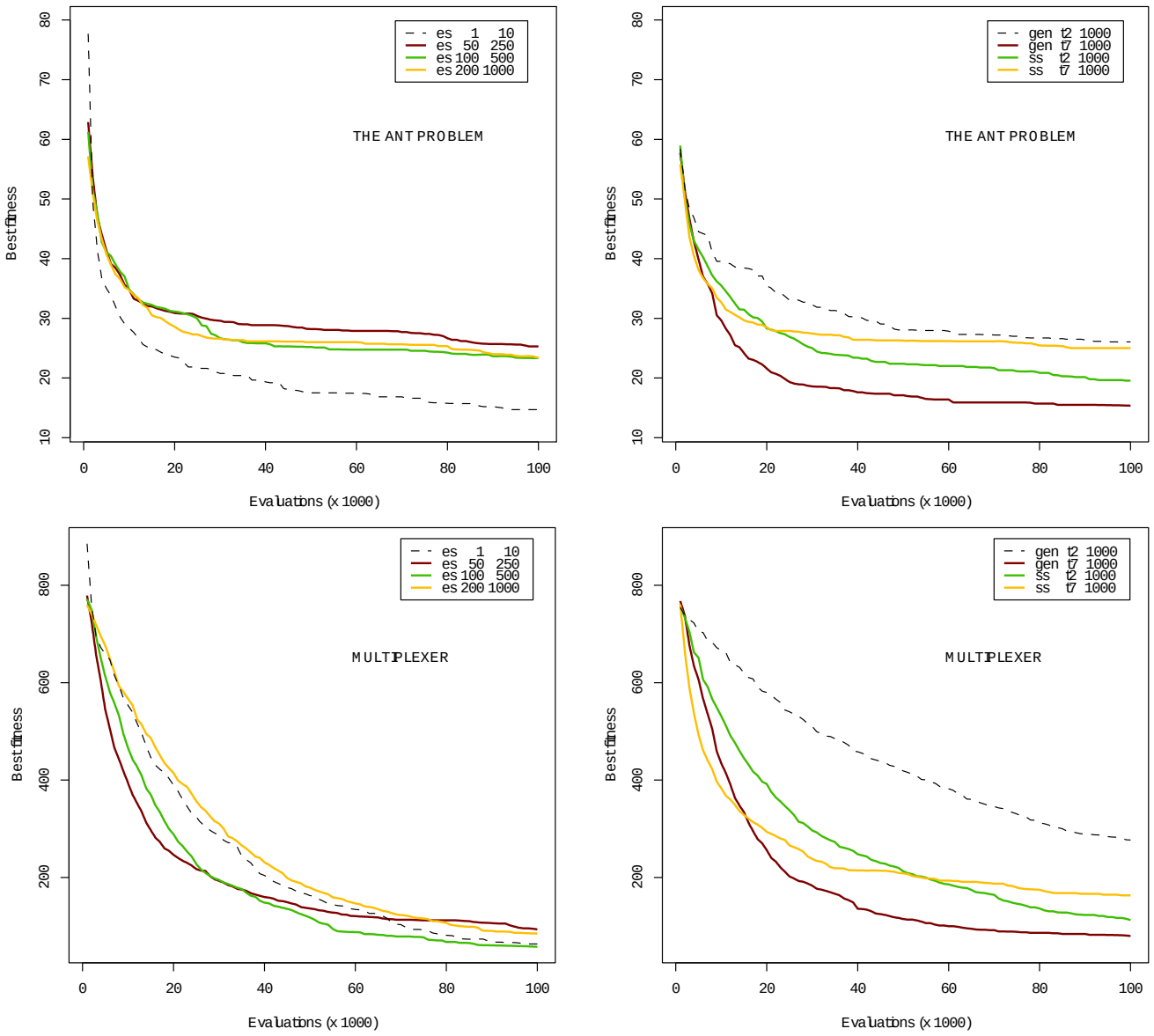

Figure 3: These graphs show the progress of the various runs. Top left shows the process of the various comma-ES runs on the ant problem, while top right shows the process of generational and steady state GP using a population of size 1000 and T=2 and T=7. The bottom graphs show the same results respectively for the 11-multiplexer problem. Results are averaged over 50 runs.

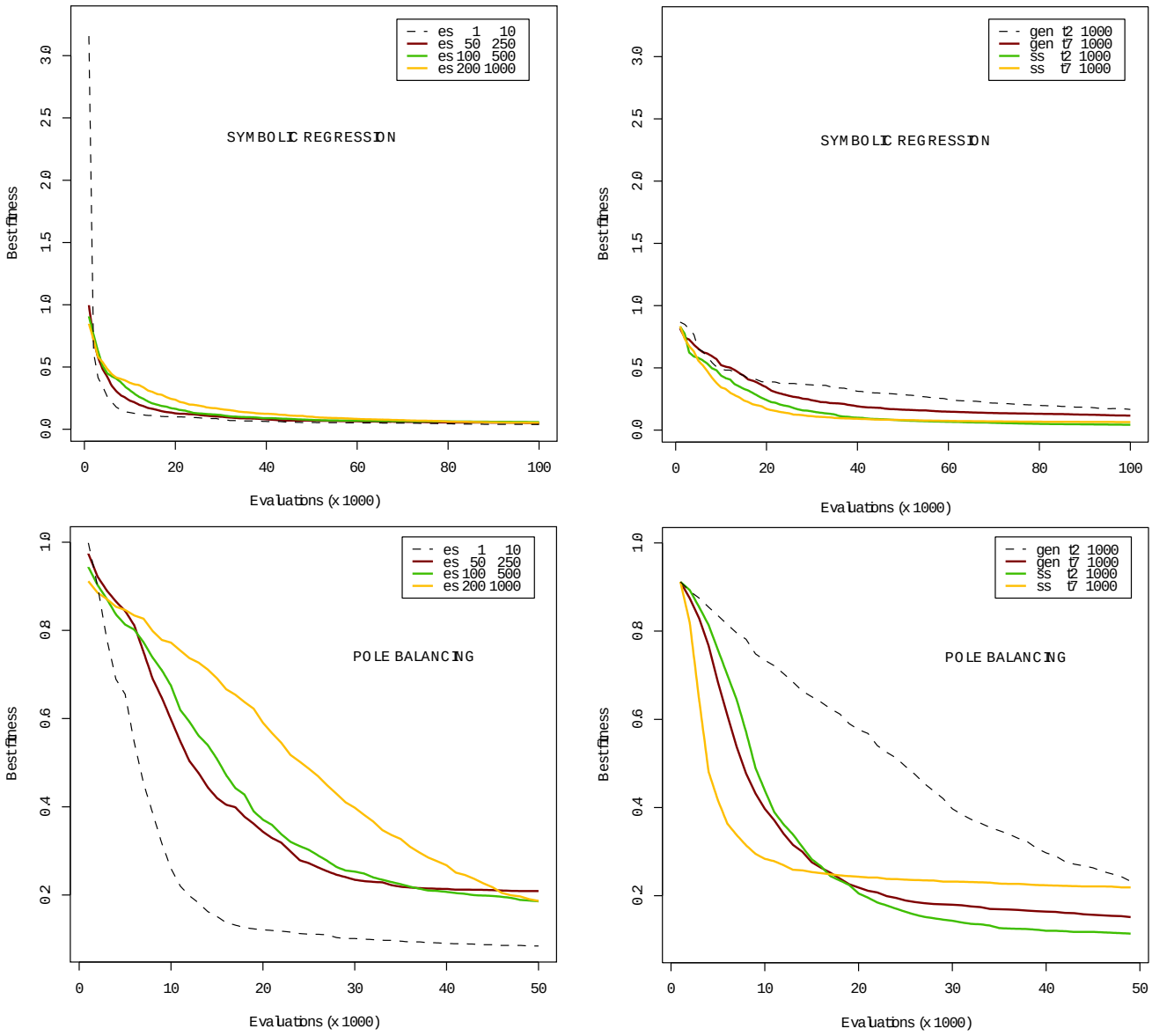

Figure 4: These graphs show the progress of the various runs. Top left shows the process of the various comma-ES runs on the symbolic regression problem, while top right shows the process of generational and steady state GP using a population of size 1000 and T=2 and T=7. The bottom graphs show the same results respectively for the pole balancing problem. Results are averaged over 50 runs.