Comparing Genetic Robustness in Generational vs. Steady State Evolutionary Algorithms

Josh Jones
Department of Computer Science
University of Idaho
Moscow, Idaho
jjones@uidaho.edu

Terry Soule[1]
Department of Computer Science
University of Idaho
Moscow, Idaho
tsoule@cs.uidaho.edu

ABSTRACT
Previous research has shown that evolutionary systems not only try to develop solutions that satisfy a fitness requirement, but indirectly attempt to develop genetically robust solutions as well - solutions where average loss of fitness due to crossover and other genetic variation operators is minimized. It has been shown that in a simple "two peaks" problem, where the fitness landscape consists of a broad, low peak, and a narrow, high peak, individuals initially converge on the lower (less fit), but broader peak, and that increasing an individual's genetic robustness through growth is a necessary prerequisite for convergence on the higher, narrower peak [18]. If growth is restricted, the population remains converged on the less fit solution. We tested whether this result holds true only for generational algorithms, or whether it applies to steady state algorithms as well. We conclude that although growth occurs with both algorithms, the steady state algorithm is able to converge on the higher peak without this growth. This result shows that the role of genetic robustness in the evolutionary process is significantly different in generational versus steady state algorithms.

Categories and Subject Descriptors
I.2.2 [**Artificial Intelligence**]: Automatic Programming, program synthesis

General Terms
Algorithms, Performance, Experimentation, Theory.

Keywords
Growth, resiliency, genetic robustness, steady-state, generational

1. INTRODUCTION
The use of genetic algorithms (GA), genetic programming (GP), and related evolutionary algorithms continues to grow as evolutionary computation becomes more widely accepted as a practical problem solving technique. However, the details of the evolutionary process and its effect on the evolution of successful solutions are not yet fully understood. Previous research has shown that although these algorithms attempt to evolve solutions that better satisfy a given fitness function, other evolutionary pressures exist as well. Specifically, it has been clearly shown that in any evolutionary system, including GP, GA, evolutionary artificial life systems, and natural biological systems, there is significant evolutionary pressure to evolve solutions that are genetically robust, in addition to being highly fit [2][17][20][21]. Genetic robustness is roughly defined as a measure of the average change in fitness of an individual as a result of genetic modification, e.g. crossover and mutation. An individual is more robust if the average change in fitness due to genetic variation is smaller.

Previous work examining the evolution of genetic robustness has largely overlooked the role of the underlying genetic algorithm, specifically whether the pressure for genetic robustness varies significantly depending on whether a generational or steady-state algorithm is used. Generational and steady state algorithms differ significantly in how individuals survive over time: how often individuals are replaced (die) versus how often they may reproduce. Because the replacement strategy is so different between generational and steady state algorithms, it is likely to have a significant effect on the role of genetic robustness. In this paper we reexamine previous results that were generated using a generational algorithm, comparing them to new results generated with a steady state algorithm. The comparison shows that the role of genetic robustness in directing the evolutionary process is significantly different in generational versus steady state algorithms.

This result has important implications for the application of evolutionary algorithms. For practical applications, where the goal is simply to find the most fit solution, it means that the decision whether to use a generational or a steady state algorithm can have a significant effect on the probability of success. Similarly, for experiments where the goal is to understand the dynamics of evolution, it means that the decision whether to use a generational or a steady state algorithm can have a significant effect on the observed outcome. The steady state results are of particular interest to the artificial life community, where attempts to model microbial evolution lead to steady state like algorithms.

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

[1] This publication was made possible by NIH Grant P20 RR16448 from the COBRE Program of the National Center for Research Resources

2. BACKGROUND

Pressure to evolve genetically robust individuals has been documented in artificial evolutionary systems in two very broad, but related ways. First, researchers have shown that under many conditions, evolving populations will converge on less fit but broader peaks in a fitness landscape, rather than on the more fit but narrower peaks. Broader peaks represent genetically robust solutions because genetic changes to individuals on those broad peaks are likely to keep the individual somewhat on the peak, i.e. genetic changes cause relatively small fitness changes, the definition of a genetically robust individual. In contrast, genetic changes to individuals on a narrow peak are likely to cause the individual to 'fall off' the peak. Second, research has shown that there are a number of strategies that evolving individuals adopt to increase their genetic robustness. These are discussed in detail below.

A number of researchers have confirmed that evolving populations may converge on less fit, but broader, fitness peaks. Wilke et al. found that with the artificial evolutionary system Avida, with sufficiently high mutation rates, individuals with low reproductive rates that occupy flatter regions of the fitness landscape will eventually replace individuals with higher reproductive rates if those individuals with high reproductive rates occupy steeper regions of the fitness landscape [21]. Wilke has also shown that in competition between quasispecies at high mutation rates, quasispecies located on a neutral network with higher connection density may out-compete quasispecies with a higher replication rate on a less connected neutral network [20]. Krakauer and Plotkin showed that small populations converge on less fit, but more robust solutions (broader peaks) whereas larger populations converge on more fit, but genetically less robust solutions (higher, narrower peaks) [6]. This previous research confirms that under the conditions of small populations or high mutational loads, populations will often converge on broader but lower fitness peaks, i.e. evolution can favor individuals with lower reproductive rates, but higher genetic robustness.

The phenomenon of code growth (or bloat) observed in genetic programming (GP) is one of the most outstanding example of the evolution of strategies to increase genetic robustness [2] [5] [7] [11] [12] [13] [17]. Code growth is a rapid increase in the size of the evolving individuals that is not directly correlated with fitness improvements. It is seen in almost every GP model, and the growth usually consists of extra code that does not contribute to the program's fitness (known as inoperative code or introns).

It is commonly accepted that the code growth occurs, at least in part, as a mechanism to increase genetic robustness, particularly with respect to crossover. This hypothesis was independently proposed by a number of researchers in 1994 [1] and 1995 [11] [12]. The general idea is that if an individual's genome is mostly inoperative code, then crossover is most likely to exchange this inoperative code, and thus not affect fitness. Therefore, by increasing the ratio of inoperative to operative code, an individual can increase its genetic robustness, which is done predominantly by increasing the amount of inoperative code.

Recent studies show that code growth is a general phenomenon, not restricted to traditional tree-based GP algorithms, nor purely inoperative code. Smith and Harries have shown that growth can occur in code that does influence fitness if the code has only a negligible effect on performance [14]. Soule et al. have shown that code growth can also occur in exons that have a significant effect on fitness [15] [19].

Additional research has shown mutation can encourage code growth in certain situations, so the phenomenon is not restricted to the crossover operator, even though the standard method of applying mutation in the evolutionary computation field appears to limits code growth [10] [19]. Theoretical research strongly suggests that code growth is a factor in any evolutionary system using a variable sized representation, regardless of the operators used [8] [9] [11].

In addition to code growth, several other strategies for increasing genetic robustness have been shown to evolve. Hogeweg and Hesper found that with high mutation rates, evolving subsequences are more likely to use "multiple coding", that is, to use existing, conserved, subsequences as recognition sites [4]. This increases robustness without necessarily increasing (or decreasing) fitness.

It has also recently been shown that in representations that include multiple 'genes' with similar effects, those genes that produce more robust individuals will tend to replace genes that produce less robust individuals [15] [16] [19]. Edlund and Adami have shown that in artificial evolutionary systems, pressure for robust individuals reduces the amount of antagonistic epistasis [3].

Recently, Soule has studied the interaction of these two phenomena: convergence on less fit, but broader peaks, and adaptation of strategies to increase genetic robustness. Using a modified generational evolutionary algorithm to solve a simple “two peaks” problem, it was shown that increasing genetic robustness could cause a population that originally converged on a less fit, broad peak to shift to a more fit, narrower peak [18]. The increasing genetic robustness of the individuals effectively 'broadened' the narrower peak making it evolutionarily competitive. In this problem, individuals are numeric strings consisting of 0s, 1s, and 4s. The strings vary in length, and have a value: the sum of all digits in the string. Fitness is a function of value, as shown in Figure 1.

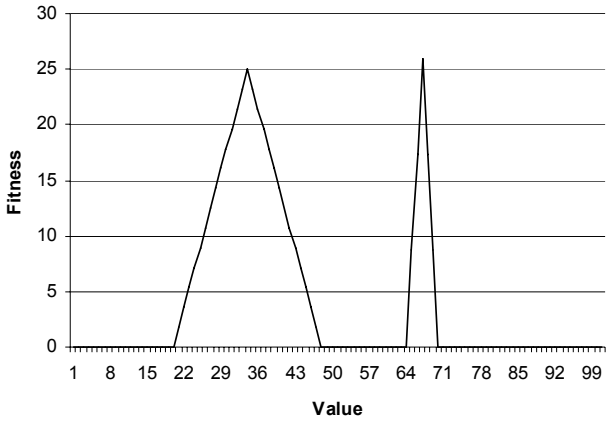

Figure 1: Individual's fitness based on its value.

Note that in Figure 1 the peak on the left is much broader, but the second peak is higher by 1 point. It was found that when using a “constant crossover” (described later), solutions initially converged on the broader peak [18]. However, as the algorithm continued, the individual strings grew in length, increasing genetic robustness. When a certain critical length was reached, individuals suddenly left

the broader peak and converged on the sharper peak. A second type of crossover, "proportional crossover", limited the individuals' growth, and the population never left the broader peak. Soule concluded that increasing genetic robustness could be a prerequisite to finding better solutions, if those solutions represented narrower fitness peaks.

The research described above used generational models of the evolutionary process, which raises the question of whether the presence and role of pressure for genetic robustness exists in all implementations of evolutionary algorithms. Specifically, would the same pressure for robustness exist in a steady state implementation? Would an increase in robustness (accomplished through growth or some other strategy) remain a necessary prerequisite to convergence on a higher, narrower peak? Given the widespread use of steady state algorithms in evolutionary computation and artificial life applications, and given the significant influence of the pressure for robustness on the evolutionary process, it is important to determine whether similar results apply to steady state algorithms.

Whereas in previous research, individuals suddenly switched from the lower peak to the higher peak when they reached a certain critical length, we hypothesize that using a steady state algorithm will cause the population to converge on a higher, narrower peak more smoothly. This hypothesis is based on the fact that in steady state algorithms the current best solutions are automatically maintained in the population – only the poorest individuals run a risk of being replaced. In contrast, in a generational algorithm every individual is replaced in every generation, thus there is a much greater pressure to produce offspring that are not degraded by crossover and mutation.

For example, consider the fate of an individual in the top 20% of the population. In a steady state algorithm, that individual will be selected as a parent fairly often and thus will have the chance to produce offspring, but it is in no danger of being replaced (at least not until the average fitness of the population rises to the point where it is no longer in the top 20% of the population). Thus, even if the individual is genetically 'brittle' and most of its offspring are significantly degraded by crossover and/or mutation, it will still survive, at least until the average fitness of the population surpasses its fitness.

In contrast, in a generational algorithm with the same selection mechanism, the same individual in the top 20% of the population will be selected to be a parent equally often, but at the end of the generation the individual is deleted along with the rest of the previous generation's population. In this case, if the individual is genetically brittle and therefore it has offspring with degraded fitness, both the offspring and the original individual are likely to die out. (Of course, it is possibly that this hypothetical individual will manage to survive into the next generation, but for the typical crossover and mutation rates used in evolutionary algorithms it is unlikely that it would escape all genetic changes for more than a few generations.)

To test our hypothesis we solved the two peaks problem using both a generational and a steady state algorithm. Although it is a small, narrow problem, we chose it due to its ability to represent a much wider variety of problems. Specifically, if it can be demonstrated that generational and steady state algorithms can produce significantly different results for this small problem, it suggests a significant difference exists between other problems as well. Additionally, the two peaks problem had previously been used to show genetic robustness was necessary to converge upon more-fit solutions[18], a result which applies to a wide variety of additional problems as well. It was necessary, therefore, to use the same problem to demonstrate the limitations of these results.

The two peaks problem is designed specifically to highlight both the preference for lower, broader peaks and the adaptation of strategies to increase robustness in various algorithms. Specifically, we measure the time required for populations evolved with each algorithm to converge on either peak, and determine whether populations evolved with the steady state algorithm converge on the higher, narrower peak more smoothly or more rapidly than those evolved with the generational algorithm. We also observed the rate of growth of the individuals, to determine whether the steady state algorithm results in as rapid code growth as the generational algorithm.

3. METHODS

The generational evolutionary algorithm was implemented identically to the one used by Soule [18], explained below. The steady state version differed only in that it was a steady state algorithm. The parameters used in these experiments are described in Table 1.

Table 1: Parameters of the Evolutionary Algorithms

Fitness	F(sum of integers in chromosome)
Integer Values	0,1,4
Population Size	500
Crossover Probability (for generational algorithm only)	90%
Mutation Probability	0% (No Mutation)
Selection	3 member tournament (1 for each parent)
Iterations	2000 generations
Maximum Length	3000 genes
Elitism (for generational algorithm only)	2 copies of the single best individual
Initial Population	Random, of lengths 5 to 59.
Number of Trials	50
Crossover	Proportional and Constant (explained below)
Peak 1 Height	25
Peak 1 Location	33
Peak 1 Width	14
Peak 2 Height	26
Peak 2 Location	66
Peak 2 Width	3

The length of the individual was allowed to vary in size, up to 3000 digits long. Solutions exceeding this length were truncated to 3000 (deleting all digits after the three thousandth.) A value was assigned to each individual by summing all the digits in its integer string. The individual's fitness was then assigned by mapping its value to a fitness using the function illustrated in figure 1. Although allowing variable length chromosomes meant the two algorithms were not standard genetic algorithms, it allowed specific advantages. First, it provided a clear measure of genetic robustness through an individual's size, and more specifically, the number of 0's in the individual's integer string. Secondly, it allowed the algorithms to more closely model GP algorithms, where genetic robustness in the form of code growth is larger topic of concern. Finally, using two different forms of crossover (explained below), we were still able to model standard genetic algorithms due to the lack of growth that resulted using proportional crossover, while still modeling GP when using constant crossover.

For the steady state algorithm, a "generation" was defined as 250 (half the population) crossovers, where each crossover generates 2 children (crossover behavior applies to both generational and steady state algorithms). Each child in the steady state algorithm then replaced an individual supplied by a worst-of-three tournament (one tournament for each child). The replacement was automatic; the child did not need to be of better fitness than the individual it was replacing. This stipulation was added because the algorithm was otherwise quick to generate enough optimal solutions to saturate the population, effectively stopping the algorithm and making it difficult to study its effects over time.

Each algorithm was tested with two different versions of crossover: proportional and constant crossovers. Proportional crossover is analogous to two-point crossover in a standard GA. Two crossover points are randomly selected on each parent, and the integer string between these two points is swapped between parents. For example:

0114|140|0104 -> 0114|104401|0104
10|104401|100 -> 10|140|100

Note that in proportional crossover, the distance between an individual's two crossover points is random, and strongly dependent on the length of the individual. The second crossover, constant crossover, developed crossover points which were not strongly dependent on the individuals' length. The first crossover point was chosen randomly, as usual. The second crossover point was picked as a function of its distance from the first crossover point. Specifically,

```
distance = 2;
while(random() < 0.50)
        distance *= 2;
crossover2 = crossover1 + distance;
```

In other words, the second crossover point was by default 2 indices ahead of the first crossover point, but that distance iteratively doubled with 50% probability. I.e. 50% of the crossover regions are of length 2, 25% are of length 4, 12.5% are of length 8, etc. The maximum allowed distance was half of the individual's length, e.g. the maximum crossed region in an individual of length N is N/2.

Constant crossover is analogous to crossover in tree-based GP. In standard tree-based GP crossover a random point is chosen for crossover. For full binary trees this results in an average crossover branch size of two nodes regardless of the tree size. I.e. larger branches are exponentially less likely to be chosen for crossover. In practice, GP usually leads to randomly shaped trees rather than full trees. However, the distribution of crossover points still heavily favors small branches and using the 90/10 rule (choosing leaf nodes for crossover only 10\% of the time) only slightly shifts the distribution towards larger branches. Thus, the distribution of crossover sizes with constant crossover is comparable to those seen in tree-based GP; both emphasize exchanging small branches.

4. Results

The generational and the steady state algorithms were tested with both types of crossovers, for a total of 4 combinations. Each combination was tested for 50 trials.

When running the algorithms, the generational algorithm performed exactly as in the original study, as expected. Figures 2 and 3 show the average value of the individuals in each generation, averaged across all 50 trials. The error bars are the standard deviation across the 50 trials. Recall that the lower (less fit), but broader peak is centered on the value 33 and the higher, narrower peak is centered on the value 66.

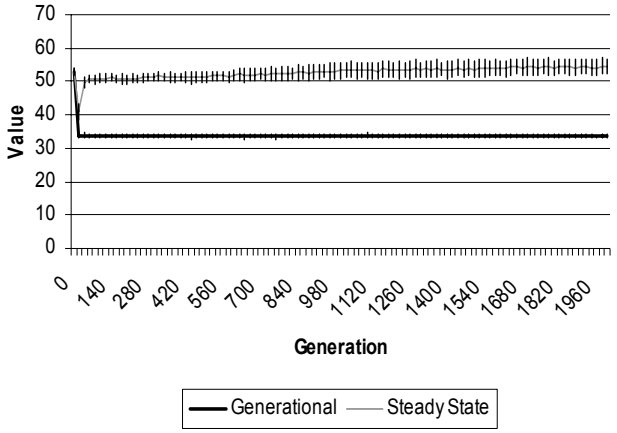

Figure 2: Average value of individuals using proportional crossover, error bars are one standard deviation.

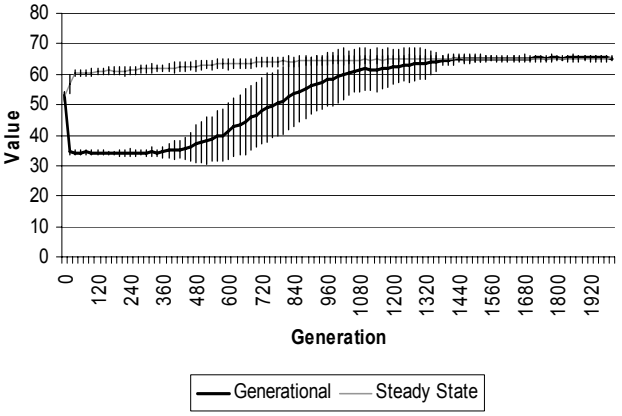

Figure 3: Average value of individuals using constant crossover, error bars are one standard deviation.

The generational algorithm using proportional crossover (Figure 2, dark line) converges on the lower peak of 33. The generational algorithm using constant crossover (Figure 3, dark line) initially converges on the lower peak, and then halfway through the run switches over to the higher peak.

The steady state algorithm produces significantly different results. Although Figure 2 (pale line) shows a brief dip to the broader peak in the very early generations, the population of the steady state algorithm, using either crossover, quickly finds the higher peak and slowly converges on it as the run continues.

Interestingly, even with the steady state algorithm the two crossovers do not behave identically. The average value of the population with constant crossover is consistently 10 higher then that of the proportional crossover. Why should this occur?

The answer also seems to be based on the evolution of genetic robustness. Because the children in the steady state algorithm replace the loser in a worst-of-three tournament, the upper 2/3rds of the population will be the more optimal solutions (solutions that are on the higher peak), while the lower 1/3rd of the population will be individuals who were produced by crossover and were potentially 'pushed' off of the higher peak. With half of all constant crossovers only changing two genes in an individual, these offspring do not differ much from their parents, and therefore are much less likely to have been pushed far from the higher peak. Thus, these offspring do not pull the average value far from the optimal. In contrast, proportional crossover causes offspring to be significantly changed from their parents, causing more variation in the individuals in the lower third of the population and a lower average value.

To help understand this behavior, two snapshots were taken of the population during one trial of proportional crossover. The snapshot counted how many individuals existed with each value. One snapshot was taken at generation 100, the other at generation 1900. The results are shown in Figures 4 and 5.

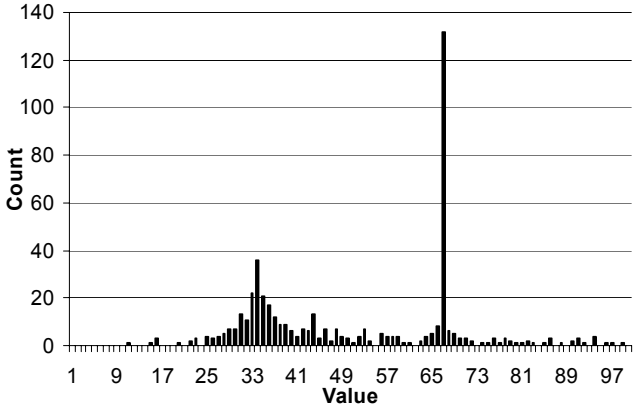

Figure 4: Distribution of individuals in the steady state algorithm, using proportional crossover, generation 100

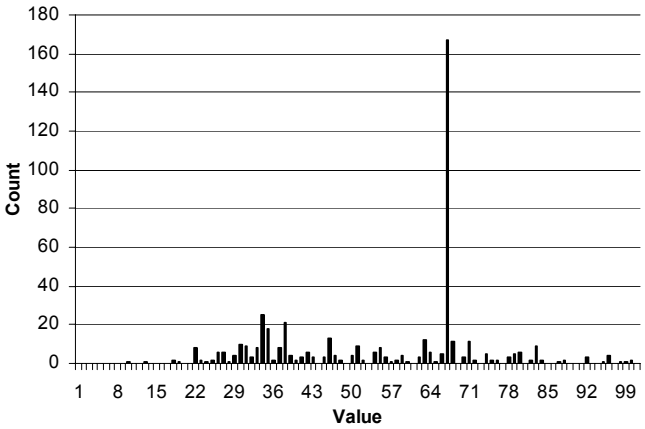

Figure 5: Distribution of individuals in the steady state algorithm, using proportional crossover, generation 1900

As expected, both figures show a large number of individuals at the higher peak, with a scattering of individuals, representing the lower ranks in the population, which bring down the average fitness. What is interesting is that these lower ranked individuals, at least initially, still tend to converge on the lower peak. It appears that the individuals on or near the lower, broader peak, have sufficient fitness to not lose the worst-of-three tournament in all circumstances, so they remained in the population, but did not have high enough fitness to be selected as a parent. As time went on, the number of individuals at the higher peak increased and the individuals at the lower peak began to lose the worst-of-three tournament more often, causing the slow increase in average value seen in Figures 2 and 3.

Next we examined the total code growth for each algorithm. It was previously shown that code growth does not occur in the generational algorithm when proportional crossover is used, but that growth does occur with constant crossover [18].

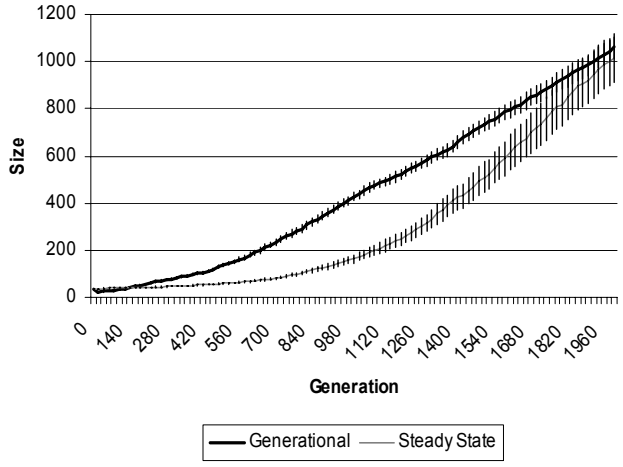

Figure 6: Average size of individuals using constant crossover, error bars are standard error.

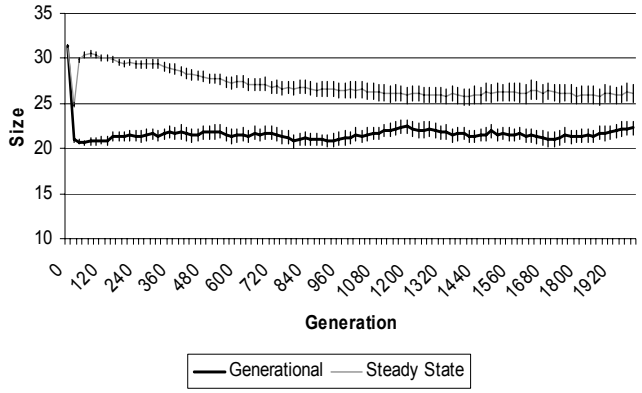

Figure 7: Average size of individuals using proportional crossover, error bars are standard error.

Figures 6 and 7 show the average individual size and standard error for constant crossover and proportional crossover with the generational and steady state algorithms. As expected, both the generational and steady state algorithms produce no code growth when proportional crossover is used. Likewise, both the generational and steady state algorithms show significant code growth when constant crossover is used. What is curious is the significant difference in the rates at which code growth occurs. The size of individuals in the generational algorithm increased steadily throughout the run. In contrast, the steady state algorithm individuals had minimal code growth throughout the first half the algorithm, followed by a rapid increase for the second half. Recall that for individuals evolved with the steady state algorithm, the average value with both constant crossover and proportional crossover increased at the same relatively slow rate (Figures 2 and 3). In particular, with constant crossover the population reached the higher average value (above the value reached when proportional crossover was used) before code growth became a factor. I.e. Figure 3 shows that with constant crossover and the steady state algorithm, the average value is constant after generation 840, which is when the rapid growth begins. Therefore, it does not seem that code growth played a role in shifting from one peak to the other with the steady state algorithm.

5. CONCLUSIONS

The results confirm that for this problem, proportional crossover limits code growth while constant crossover allows it, even with a steady state algorithm. This supports the theory that growth only occurs when the number of 'genes' changed by crossover (and presumably other genetic variation operators) is independent of individuals' total size, regardless of whether a generational or steady state algorithm is used. I.e. if the number of genes changed by genetic operators is proportional to the size of the evolving individuals, growth is no longer an effective strategy to increase robustness and does not occur. The results also demonstrate that both the rate of growth and, more importantly, the growth curve, are functions of the algorithm used.

More importantly, the results show that code growth with the steady state algorithm, and presumably robustness in general, is not a prerequisite to the population's shifting to a narrower peak in the fitness landscape. With the steady state algorithm and either form of crossover, the population shifted to the higher, narrower peak within a few generations, and the average fitness with both constant and proportional crossover are very similar, even though one shows rapid growth and the other shows no growth. In contrast, code growth in the generational algorithm, or presumably some other mechanism to increase robustness, is a necessary prerequisite to the population's shifting to the higher, narrower peak. Thus, these results show that the role of genetic robustness is significantly different in steady state, as opposed to generational, evolutionary algorithms. Given our increasing understanding of the importance of pressure for genetic robustness in influencing the tempo and dynamics of evolution, it is important to understand exactly how this pressure differs between the two algorithms.

In particular this research suggests several significant questions that need to be addressed. In what other ways do the performance of a generational and steady state algorithm differ? How important are these potential differences to other research results? Is code growth a significant hindrance in genetic programming, or does it serve a necessary purpose?

6. REFERENCES

[1] Blickle, T., and Thiele, L. Genetic Programming and Redundancy. In *Genetic Algorithms within the Framework of Evolutionary Computation.* Max-Planck-Institut fur Informatik, Saarbrucken, Germany, 1994, 33-38.

[2] DeVisser, A.G.M., Hermission, J., Wager, G., Meyers, L.A., Bagheri-ChaiChain, H., Blanchard, J.L., Chao, L., Cheverud, J.M., Elena, S.F., Fontana, W., Gibson, G., Hansen, T.F., Krakauer, D., Lewontin, R.C., Ofria, C., Rice, S.H., von Dassow, G., and Wagner, A. Perspective: Evolution and Detection of Genetic Robustness. *Evolution, 57*, 2003, 1959-1972.

[3] Edlund, J., and Adami, C. Evolution of Robustness in Digital Organisms. *Aritficial Life,* 10, 2004, 167-179.

[4] Hogeweg, P., and Hesper, B. Evolutionary Dynamics and the Coding Structure of Sequences: Multiple Coding as a Consequence of Crossover and High Mutation Rates. *Computers and Chemistry, 16,* 1992, 171-182.

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

[6] Krackauer, D.C., Plotkin, J.B. Redundnacy, antiredundancy, and the robustness of genomes. *PNAS,* 99 (2002), 1405-1409.

[7] Langdon, W. *Fitness Causes Bloat: Simulated Annealing, Hill Climbing and Populations.* Technical Report CSRP-97-22, University of Birmingham, Birmingham, UK, 1997.

[8] Langdon, W., and Poli, R. Fitness Causes Bloat. In *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing,* Springer-Verlag, London, 1997, 13-22.

[9] Langdon, W., and Poli, R. Fitness Causes Bloat: Mutation. In *Proceedings of the First European Workshop on Genetic Programming,* Springer-Verlag, Paris, 1998, 37-48.

[10] Langdon, W., Soule, T., Poli, R., and Foster, J. The Evolution of Size and Shape. In *Advances in Genetic Programming III.* The MIT Press, Cambridge, MA, 1999, 163-190.

[11] McPhee, N.F., and Miller, J.D. Accurate Replication in Genetic Programming. In *Proceedings of the Sixth International Conference on Genetic Algorithms.* Morgan Kaufmann, San Francisco, CA, 1995, 310-317

[12] Nordin, P., and Banzhaf, W. Complexity Compression and Evolution. In *Proceedings of the Sixth International Conference on Genetic Algorithms.* Morgan Kaufmann, San Francisco, CA, 1995, 310-317.

[13] Nordin, P., Banzhaf, W., and Francone, F. Introns in Nature and in Simulated Structure Evolution. In *Proceedings Bio-Computing and Emergent Computation.* Springer, 1997, 22-35.

[14] Smith, P., and Harries, K. Code Growth, Explicitly Defined Introns, and Alternative Selection Schemes. *Evolutionary Computation, 6,* 4 (1998), 339-360

[15] Soule, T. Exons and Code Growth in Genetic Programming. In *Genetic Programming, 5th European Conference, EuroGP 2002,* 2002, 142-151.

[16] Soule, T. Operator Choice and the Evolution of Robust Solutions. In *Genetic Programming Theory and Practice,* Riolo R., and Worzel, B., 2003, 257-270.

[17] Soule, T. Resilient Individuals Improve Evolutionary Search. *Artificial Life, 12,* 1 (Winter 2005), 17-34.

[18] Soule, T., Foster, J., and Dickinson, J. Code Growth in Genetic Programming. *In Genetic Programming 1996: Proceedings of the First Annual Conference.* MIT Press, Cambridge, MA, 1996, 215-223.

[19] Soule, T., Heckendorn, R., and Shen, J. Solution Stability in Evolutonary Computation. In *Proceedings of the 17th International Symposium on Computer and Information Systems*, 2002, 237-241.

[20] Wilke, C. Selection for Fitness versus Selection for Robustness in RNA Secondary Structure Folding. *Evolution,* 55 (2001), 2412-2420.

[21] Wilke, C., Wang, J.L., Ofria, C, Lenski, R.E., and Adami, C. Evolution of digital organisms at high mutation rates leads to survival of the flattest. *Nature,* 412 *(*2001), 331-33.