A Comparison of Crossover and Mutation in Genetic Programming

Sean Luke
seanl@cs.umd.edu
http://www.cs.umd.edu/~seanl/

Department of Computer Science
University of Maryland
College Park, MD 20742

Lee Spector
lspector@hampshire.edu
http://hampshire.edu/~lasCCS/

School of Cognitive Science and Cultural Studies
Hampshire College
Amherst, MA 01002

ABSTRACT

This paper presents a large and systematic body of data on the relative effectiveness of mutation, crossover, and combinations of mutation and crossover in genetic programming (GP). The literature of traditional genetic algorithms contains related studies, but mutation and crossover in GP differ from their traditional counterparts in significant ways. In this paper we present the results from a very large experimental data set, the equivalent of approximately 12,000 typical runs of a GP system, systematically exploring a range of parameter settings. The resulting data may be useful not only for practitioners seeking to optimize parameters for GP runs, but also for theorists exploring issues such as the role of "building blocks" in GP.

1 Introduction

The relative merits of crossover, mutation, and other genetic operators have long been debated in the literature of genetic algorithms. The traditional view is that crossover is primarily responsible for improvements in fitness, and that mutation serves a secondary role of reintroducing alleles that have been lost from the population. This view is consistent with the notion that evolutionary progress is best made by combining building blocks (or *schemata*) from high-fitness individuals (see for example [Holland 1975]).

But the roles of building blocks and of crossover have become increasingly controversial in recent years. Several researchers have presented new theoretical arguments and empirical results showing that mutation can be more useful than was previously thought (for example, [Shaffer and Eshelman 1991; Tate and Smith 1993; Hinterding, Gielewski and Peachey 1995]). Others have produced new arguments in favor of crossover (for example [Spears 1993]).

The mutation/crossover debate has produced a variety of insights about the nature of genetic algorithms, and there is more yet to be discovered. Mitchell concludes:

> ... it is not a choice between crossover or mutation but rather the balance among crossover, mutation, and selection that is all important. The correct balance also depends on details of the fitness function and the encoding. Furthermore, crossover and mutation vary in relative usefulness over the course of a run. Precisely how all this happens still needs to be elucidated. [Mitchell 1996, p. 174]

For genetic programming (GP), the issue is even less resolved, and the lessons to be learned from a systematic study of the issue may be correspondingly more interesting. Crossover in GP swaps subtrees that may be of different sizes and in different positions in their programs, and mutation in GP generates entire new subtrees—both of these operations are so different from their traditional counterparts that one might be surprised if the analysis or results from traditional genetic algorithms hold for GP at all. The issue has been briefly addressed in previous GP work—for example, [Koza 1992] presented results from one problem and argued that mutation has little utility in GP—but we will argue that the conclusions in previous work were based on insufficient data.

Koza has argued that mutation is in fact useless in GP because of the position-independence of GP subtrees, and because of the large number of chromosome positions in typical GP populations [Koza 1992, pp. 105–107]. He has also published data for a problem (Boolean 6-multiplexer) that supports this argument [Koza 1992, pp. 599–600], and he uses no mutation in the bulk of his work. Others in the field seem to take a similar position; most published GP research uses either no mutation or small mutation rates.[1]

The received wisdom notwithstanding, the data presented in this paper show that mutation can in fact have utility, and that crossover does not consistently have a considerable advantage over mutation. As a result, previous arguments against mutation may be based on premises that, while plausible, do not actually hold across GP domains.

[0]To Appear in: *Proceedings of the Second Annual Conference on Genetic Programming (GP-97)*, J. Koza *et al.*, editors. Morgan Kaufmann, 1997.

[1]Although there are exceptions; for example [Banzhaf, Frankone and Nordin 1996], although this work uses linear genotypes.

But the data is also complex, pointing not to simple revision of advice about parameter values, but rather to a range of new open questions. A better understanding of the factors that produce this data should therefore enhance our understanding of the fundamental nature of genetic operators in GP. There has been considerable recent interest both in adapting GP crossover and mutation to non-GP selection mechanisms [O'Reilly and Oppacher 1996] and in adopting exotic new genetic operators for GP [Angeline 1996, Iba and de Garis 1996, Teller 1996]. In light of these recent investigations, a firmer understanding of the roles of standard crossover and mutation in GP would certainly seem to be timely.

The bulk of this paper is devoted to the simple presentation of the data from our large, systematic series of runs (the equivalent of approximately 12,000 typical runs of a GP system). Following the descriptions of the runs and the presentation of the results we briefly highlight the more obvious lessons from the data—mainly that the underlying story is more complex than one might have guessed—and provide a few speculations about pieces of the underlying story. We conclude with a challenge to other researchers to explain intriguing nonlinear features of the data.

2 Runs

Our runs are divided into two sets. In our first set of runs, we compared an all-crossover approach with an all-mutation approach to the problem, examining where each had advantages, and looking for a "break-even point" beyond which one approach began to be consistently more successful. In our second set of runs, we compared various blends of mutation and crossover over a variety of population sizes, in an attempt to ascertain how much crossover was really beneficial.

One of the difficulties in comparing features in Genetic Programming is the large number of external parameters which can bias the results. To cope with this, we identified the four parameters we thought would have the most dramatic bias on our data. We performed runs with a broad variety of settings for these parameters:

- **Problem domain.** We picked four different domains from the literature: two "easy" problems (6-Multiplexer and Lawnmower), and two "harder" problems (Symbolic Regression and Artificial Ant). Of course, problem difficulty is not absolute; we describe these domains as "easy" or "harder" only in relation to other domains in the study.

- **Population size.** Depending on domain, we tested population sizes ranging from 8 to 2048.

- **Number of generations.** Depending on domain, we performed runs up to 512 generations. After each generation, we noted the success up to that point, giving us data for a variety of numbers of generations.

- **Selectivity.** We chose to run all four domains using tournament selection, because it allowed us to rigorously vary selectivity simply by changing the tournament size. In each domain, we used two different tournament sizes: 2 (because it is the standard in GA literature, and because it is not very selective) and 7 (because it is used extensively in GP literature, and also is relatively highly selective).

The number of parameter combinations was large, requiring a correspondingly large number of runs. Further, each "run" shown in the data and figures is actually the average of 25 random runs with the same set of parameters. All told, the data in the paper is the result of 572,947,200 evaluations, or the equivalent of about 12,000 runs of typical size in the GP community (say, 50 generations, population size 1000). We performed runs using *lil-gp 1.02* [Zongker and Punch 1995], running on a 40-node DEC Alpha workstation cluster. No animals were killed or injured during the procedure.

Other than changing the selection scheme (to tournament selection), we tried to keep our default parameters for all domains as close to the traditional parameters as possible. For all of our runs, we chose to include 10% reproduction, in order to stay closer to the classic GP mix as outlined in [Koza 1992]. We imposed a maximum tree depth limit of 17. We used a depth ramp of between 2 and 6 for initial tree generation, and between 1 and 4 for subtree mutation. Subtree mutation picked internal nodes 90% of the time and external nodes 10% of the time. For both initial tree generation and subtree mutation, we used half-GROW, half-FULL tree-generation. Our runs did not stop prematurely when a 100% correct individual was found, but continued until each run was completed.

The function sets and evaluation mechanisms for the Ant, Regression, and 6-Multiplexer domains were those outlined in [Koza 1992]. The Artificial Ant domain used the "Santa Fe" trail, and allowed the ant to move up to 400 times. The target function for the Symbolic Regression domain was $x^4 + x^3 + x^2 + x$. Our implementation of the Symbolic Regression domain used no ephemeral random constants. The function set, evaluation mechanism, and tree layout (with two Automatically Defined Functions or ADFs) for the Lawnmower domain are given in [Koza 1994], using an 8x8 lawn.

We used standard "point" mutation (in which a random subtree is replaced with a new random tree) as described by [Koza 1992, p. 106]. Although other mutation techniques have also been described in the literature, consideration of these techniques is beyond the scope of this study. Similarly, we used the traditional GP crossover and reproduction operators described in [Koza 1992].

3 Comparing 90% Crossover with 90% Mutation

We began by comparing a 90% crossover, 10% reproduction scheme with a 90% mutation, 10% reproduction scheme. We compared runs under all four domains and both tournament-size options (2 and 7).

For the two "harder" domains (Symbolic Regression and Ant), we performed runs up to 512 generations long, for population sizes 64, 128, 256, 512, 1024, and 2048. Since the "easy" domains (6-Multiplexer and Lawnmower) achieved success much more rapidly and with smaller populations, for these domains we performed runs up to 64 generations long, for population sizes 16, 32, 64, 128, 256, and 512. The resultant fitnesses were the mean standardized fitness of the best-of-run individual, averaged over 25 random runs. The comparison graphs are black where crossover is better than mutation, white where mutation is better than crossover, and gray where the difference between the two is statistically insignificant (using a two-sample *t*-test at 95%).

Our results are shown in Figures 1 through 4. Before analyzing the results, some caveats: first, note that the graphs are linear in number of generations but logarithmic in population size. As a result, while at first glance it appears that population size is the predominant factor in determining fitness (the graphs often appear "flatter" with respect to number of generations), much of this is due to the the logarithmic scaling of population size. Second, note that the fitness metric shown in these graphs is the mean standardized fitness, which is monotonic but not usually linear (depending on domain). A doubling in standardized fitness does not necessarily translate to some doubling in "real fitness", if such a thing can be gauged. Lastly, remember that the scaling, both in population size and number of generations, is smaller for the "easy" problems.

From these results we concluded:

- Crossover was more successful in the majority of tests.

- Crossover tended to cause fitness to rise more rapidly, often doing better early-on and with larger populations. Mutation tended to do better with smaller populations and more generations. This was especially true for the problems with "smoother" graphs (Regression and 6-Multiplexer). The large exceptions to this rule were the in the Artificial Ant domain, where mutation would often do well early-on, and in the Lawnmower domain (Tournament size 7), where mutation did unusually well in the upper-right quadrant of the graph.

- Although mutation performed better in several runs, it appears that crossover tended to perform better where it counts: situations in which the runs were resulting in higher fitness values overall.

- Despite the conclusions above, there were *very few cases* in which there was a *large* difference between an all-crossover or all-mutation strategy. In fact, for some domains (especially Lawnmower, tournament size 7) much of the difference between the two was statistically insignificant. And in most situations, changing selectivity made a larger difference than picking crossover over mutation.

While these graphs give some idea of where crossover and mutation are better than one another with respect to population size and number of generations, they do not provide a clear picture of the relative merits of crossover and mutation in terms of computational effort. To examine this further, we compared strategies with respect to the total number of evaluations necessary to perform a run.

We first grouped <*PopulationSize, NumGenerations*> tuples into *evaluation classes*; runs in the same evaluation class required *approximately* the same number of evaluations to run. We assumed that runs requiring *n* evaluations should be in the same class as (or no more than one class below) runs requiring 2*n* evaluations. Accordingly, we grouped evaluations into classes delimited by powers of two. A tuple is in evaluation class *e* (an integer) where $e = \lfloor \log_2(\textit{PopulationSize} \times \textit{NumGenerations}) \rfloor$. The two "easy" domains cover 12 classes (4 through 15), grouping runs from 16 to 32,768 evaluations long. The two "hard" cover 15 classes (6 through 20), grouping runs from 64 to 1,048,576 evaluations long.

Once we had grouped our graph coordinates into evaluation classes, we then compared 90% mutation and 90% crossover results by class for both 2 and 7 tournament sizes. Our results are shown in Figure 5. For each domain, we show the best fitness achieved by any run in the entire class, and the average "best" fitness for runs in that class.

From these results we concluded that crossover tends to result in the highest fitness by number of evaluations. Additionally, in most cases the average fitness achieved with crossover for an evaluation class was better than the average fitness achieved by mutation. However, we note that once again the differences, were not very large. In many cases the difference between the two was not even statistically significant. Again, picking the right tournament size often made a larger difference than selecting crossover over mutation.

4 Comparing Various Combinations of Crossover and Mutation

The second question we aimed to address was: in areas where neither all-crossover nor all-mutation have a decisive advantage, is there some blend of the two which might achieve the best results? To test this, we ran various combinations of crossover over mutation for each of the four domains, using a tournament size of 7. For each domain, we chose a "typical" number of generations for a run of that domain, namely, $\frac{1}{8}$ of the number of runs we had done for the all-crossover-vs.-all-mutation experiments. This came to 8 generations for the two "easy" domains, and 64 generations for the two "hard" domains. Conveniently, these numbers of generations cut across areas in the all-crossover-vs.-all-mutation graphs where there was not a clear-cut difference between crossover and mutation. For these runs we used 10% reproduction and varied the percent of mutation from 0% to 90%, with crossover taking up the slack.

Our results are shown in Figure 6. The graphs shown in these figures are the best and average fitnesses of the population after the "typical" number of generations, for various population sizes and percentages of mutation versus crossover.

For a "typical" number of generations, we detected no significant advantage to any particular combination of crossover and mutation; all performed about the same. We did note a very slight trend towards better results as the amount of crossover increased. This peaked at around 20–30% mutation (that is, 60–70% crossover—the remaining 10% is reproduction). Interestingly, in few situations did a total crossover solution perform better than any others. In fact, in one domain (Ant), it usually came in dead last.

5 Discussion

The common wisdom of the GP community has favored a pure-crossover approach. However, our results lead us to conclude that while crossover does often yield better results than subtree mutation, the difference between the two is usually not very significant. This forces us to reevaluate the common wisdom, and to speculate a little on why crossover is not performing as well as one might expect.

Crossover in Genetic Programming is most touted for its ability to spread valuable features from one individual to another, an idea borrowed from the *Building Block Hypothesis* or *Schema Theorem* of traditional Genetic Algorithms [Holland 1975]. The validity of the Building Block Hypothesis is still a source of fierce debate within the Genetic Algorithms community [Mitchell 1996, p. 125]. But even if one assumes its validity for traditional GAs, there are key differences between GAs and Genetic Programming that argue against its application to GP (for example, [O'Reilly and Oppacher 1995]).

One important difference is that crossover in GP swaps trees of varying sizes, shape, and position, whereas the traditional Genetic Algorithm swaps alleles at exactly the same locus. A second, less-recognized difference is that Genetic Programming is used primarily to evolve *computer programs*, for which the "linkage problem" discussed in the GA literature (for example, [Mitchell, p. 159]) is particularly severe. The functions and terminals at work in GP programs are not independent of each other but instead are usually closely linked (as in any algorithm) through functional, control, and data *dependencies*.

- A *functional dependency* exists between a child and its parent when the data passed from one to the other affects the operation or result of either. Changing a child could dramatically alter the parent's operation (and vice versa).

- A *control dependency* exists when changing a subtree changes the flow of control in the program, affecting whether or not a sibling subtree is executed at all.

- A *data* or *domain dependency* exists when functions and terminals write to a shared memory mechanism [Teller 1994], or take turns manipulating a global environment in an explicit order. Changing a function or terminal can have a dramatic effect on the operations of other functions and terminals not only within its own subtree but *throughout an individual*.

Crossover between individuals does not just break a relationship between a subtree and its parent node; it also can break many global dependencies between operators in the tree, and it can modify or introduce completely different global dependencies in the second individual when this subtree is added. Depending on the domain, crossover can change the operation of the individual in dramatic ways not limited to the local area surrounding the crossover point itself. In a similar vein, crossover can "turn on" or "turn off' the several kinds of dependency-caused introns discussed in [Andre and Teller, 1996].

We speculate that the noise caused by breaking and creating the dependencies inherent in Genetic Programming may be drowning out much of benefit crossover ordinarily would provide in terms of transferring "value" from one individual to another. Depending on domain, a crossed-over subtree may be so dependent on global dependencies for its operation that its introduction into a new individual dramatically changes its previous local effect. In the very worst case, where a domain would tend to create complex webs of global dependencies, crossover and mutation may be both acting as little more than randomization operators.

6 Conclusions

The data we have presented in this paper is from a large number of runs over a wide range of parameters, presenting a broader set of experiments than given in much of the GP literature to date. Our analysis of the data indicates that crossover is more successful than mutation overall, though mutation is often better for small populations, depending on the domain. However, the difference between the two is usually small, and often statistically insignificant. Further, no particular combination of the two seems to consistently perform significantly better than either alone.

We hope that the data from this systematic comparison may be useful for those interested in optimizing GP domains, in analyzing the relative merits of crossover and mutation in Genetic Programming, and in debating the role of (or existence of) "building blocks" in GP. While we have offered our own initial speculations as to the results of this study, we challenge the GP community to find a (likely better) explanation of these results in light of current trends and theoretical results.

Of particular interest may be the interesting nonlinearities in some graphs of our data. For example, crossover for the Artificial Ant problem (tournament size 7) does unusually poorly at large population sizes relative to mutation. This is quite unlike other areas of the graph; only after about 120 generations (much longer than a typical run) does crossover become more useful. Another example is wide disparity in fitness for various combinations of crossover and mutation at large population sizes in the Artificial Ant domain. Why do some combinations (90%, 70%, and 10% mutation, for example) do so much better than their neighbors (30%, 60% mutation, etc.) at a population size of 2048? Our first thought on producing some of these graphs (in particular, those for the Artificial Ant and Lawnmower domains) was that we were seeing noise. But the number of runs conducted for each data point and the statistical significance test underlying the comparison graphs suggest otherwise. We believe that it may be instructive to further analyze the cause of these nonlinearities. In lieu of more complete explanations, we may provisionally conclude that interpreters of smaller-scale studies may be easily misled by such nonlinearities, and that the standards for the scale of comparative GP studies must be correspondingly increased.

Acknowledgements

This research was supported in part by grants from ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), and ARPA contract DAST-95-C0037. Thanks also to Jim Hendler for his help in the development and preparation of this paper, and to our reviewers for their helpful ideas and suggestions.

Bibliography

Andre, D. and Teller, A. 1996. A Study in Program Response and the Negative Effects of Introns in Genetic Programming. In *Proceedings of the First Annual Conference on Genetic Programming (GP96),* edited by John Koza et al. The MIT Press. pp. 12–20.

Angeline, P.J. 1996. Two Self-Adaptive Crossover Operators for Genetic Programming. In *Advances in Genetic Programming 2,* edited by P.J. Angeline and K.E. Kinnear, Jr. The MIT Press. pp. 89–109.

Banzhaf, W., F.D. Francone, and P. Nordin. 1996. The Effect of Extensive Use of the Mutation Operator on Generalization in Genetic Programming Using Sparse Data Sets. In *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation,* edited by H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel. Springer Verlag. pp. 300–309.

Hinterding, R., H. Gielewski, and T.C. Peachey. 1995. The Nature of Mutation in Genetic Algorithms. In *Proceedings of the Sixth International Conference on Genetic Algorithms,* edited by L.J. Eshelman. Morgan Kaufmann. pp. 65–72.

Holland, J. H. 1975. *Adaption in Natural and Artificial Systems.* University of Michigan Press.

Iba, H., and H. de Garis. 1996. Extending Genetic Programming with Recombinative Guidance. In *Advances in Genetic Programming 2,* edited by P.J. Angeline and K.E. Kinnear, Jr. The MIT Press. pp. 69–88.

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

Koza, J.R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* The MIT Press.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms.* The MIT Press.

O'Reilly, U.-M., and F. Oppacher. 1995. The Troubling Aspects of a Building Block Hypothesis for Genetic Programming. In *Foundations of Genetic Algorithms 3,* edited by L.D. Whitley and M.D. Vose. Morgan Kaufmann. pp. 73–88.

O'Reilly, U.-M., and F. Oppacher. 1996. A Comparative Analysis of Genetic Programming. In *Advances in Genetic Programming 2,* edited by P.J. Angeline and K.E. Kinnear, Jr. The MIT Press. pp. 23–44.

Shaffer, J.D., and L.J. Eshelman. 1991. On Crossover as an Evolutionarily Viable Strategy. In *Proceedings of the Fourth International Conference on Genetic Algorithms,* edited by R.K. Belew and L.B. Booker. Morgan Kaufmann. pp. 61–68.

Spears, W.M. 1993. Crossover or Mutation? In *Foundations of Genetic Algorithms 2,* edited by L.D. Whitley. Morgan Kaufmann.

Tate, D.M., and A.E. Smith. 1993. Expected Allele Coverage and the Role of Mutation in Genetic Algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms,* edited by S. Forrest. Morgan Kaufmann. pp. 31–37.

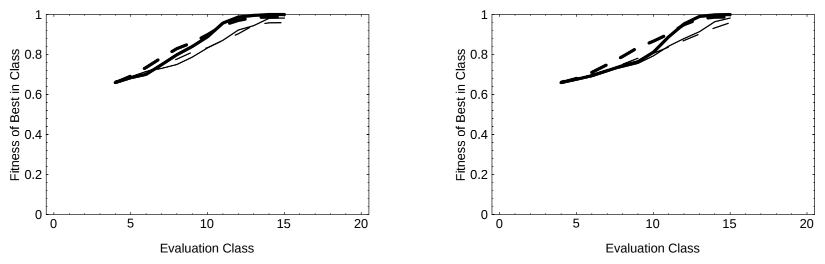
Teller, A. 1994. The Evolution of Mental Models. In *Advances in Genetic Programming,* edited by K.E. Kinnear Jr. pp. 199-219. Cambridge, MA: The MIT Press.

Teller, A. 1996. Evolving Programmers: The Co-evolution of Intelligent Recombination Operators. In *Advances in Genetic Programming 2,* edited by P.J. Angeline and K.E. Kinnear, Jr. The MIT Press. pp. 45–68.

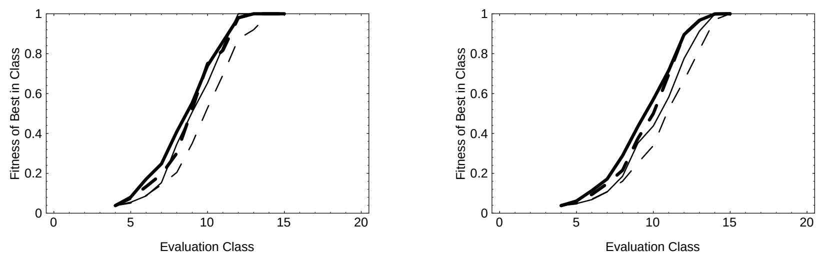
Zongker, D., and B. Punch. 1995. *lil-gp 1.0 User's Manual.* Available through the World-Wide Web at http://isl.cps.msu.edu/GA/software/lil-gp, or via anonymous FTP at isl.cps.msu.edu in the /pub/GA/lilgp directory.

Best Fitness in Entire Class **Average of Best Fitness for Runs in Class**

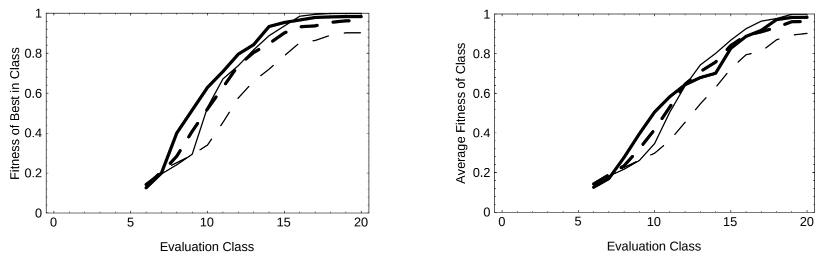
6-Multiplexer



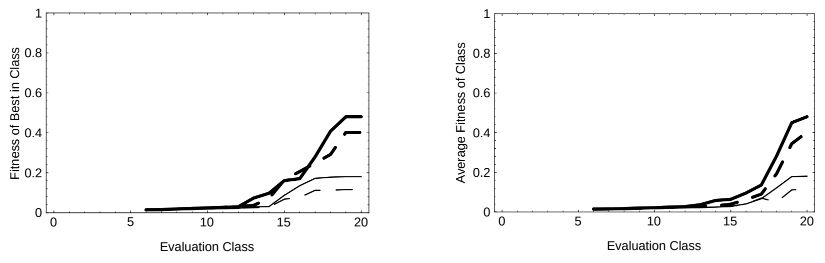
Lawnmower



Regression



Ant



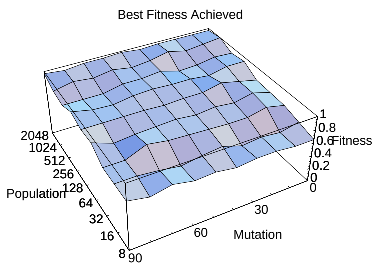
Legend

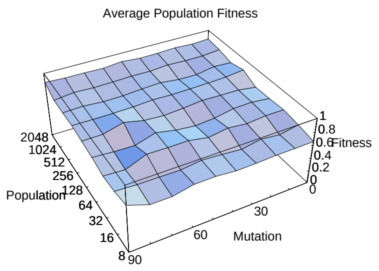
7-Tournament Crossover	7-Tournament Mutation	2-Tournament Crossover	2-Tournament Mutation
━━━━━━	━ ━ ━ ━	──────	— — — —

Figure 5. Best-of-Class Graphs. "Best Fitness in Entire Class" is the best fitness achievable by a run in an evaluation class, given an ideal combination of population size and number of generations. "Average of Best Fitness for Runs in Class" is the typical best fitness achieved by members of the class.

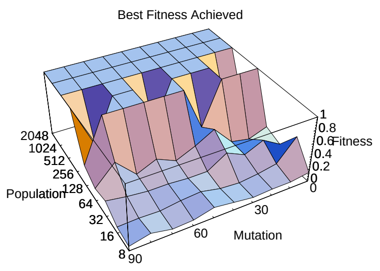
Best Fitness Achieved **Average Population Fitness**

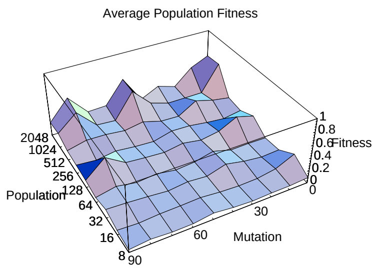
6-Multiplexer *8 Generations*



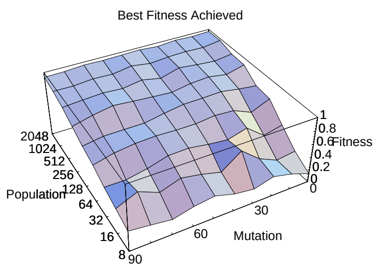


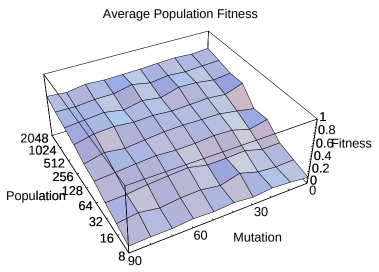
Lawnmower *8 Generations*



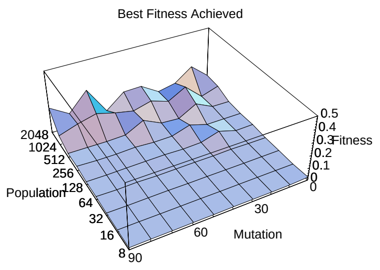


Symbolic Regression *64 Generations*





Ant *64 Generations*



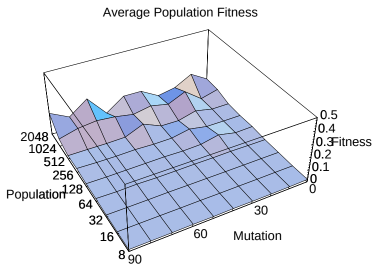


Figure 6. Fitness values after the specified number of generations, for various percentages of mutation versus crossover. In all cases, runs included 10% reproduction.

Crossover **Mutation** **Comparison**

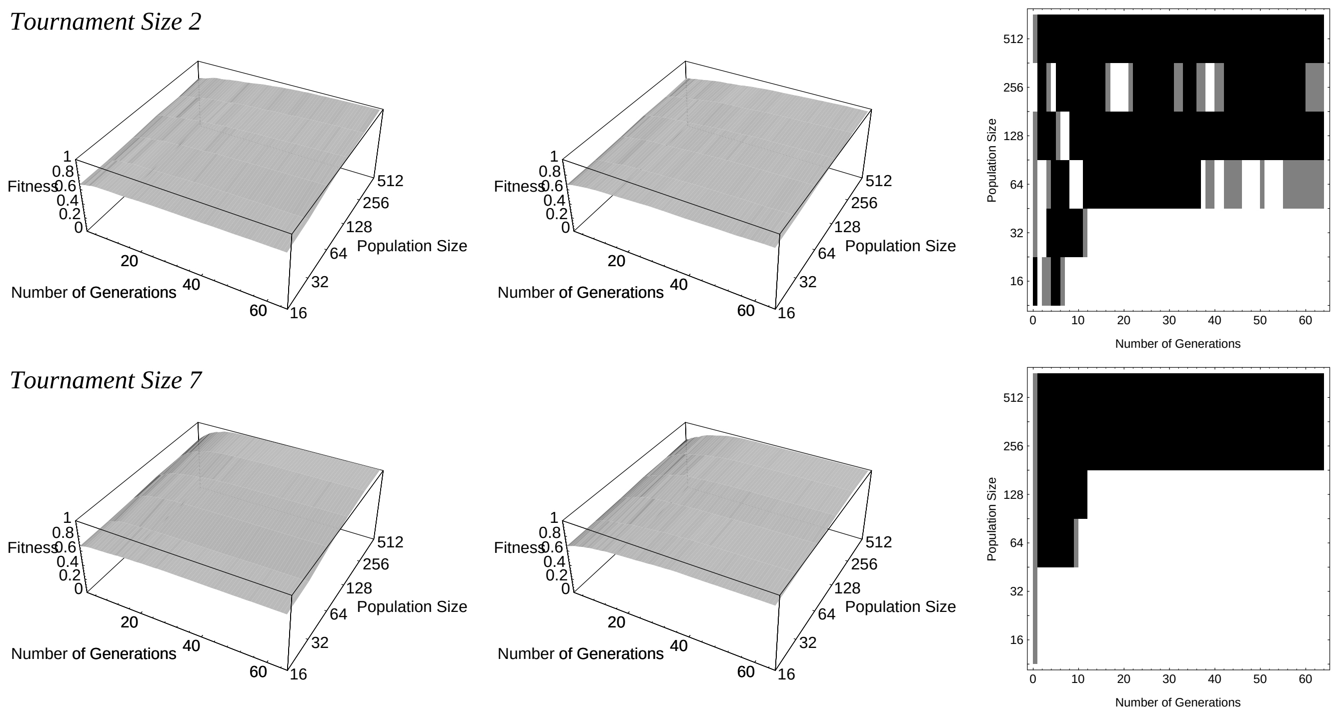


Figure 1. Comparison of crossover and mutation for the 6-Multiplexer problem. Comparison graphs are black where crossover wins, white where mutation wins, and gray where the difference is insignificant.

Crossover **Mutation** **Comparison**

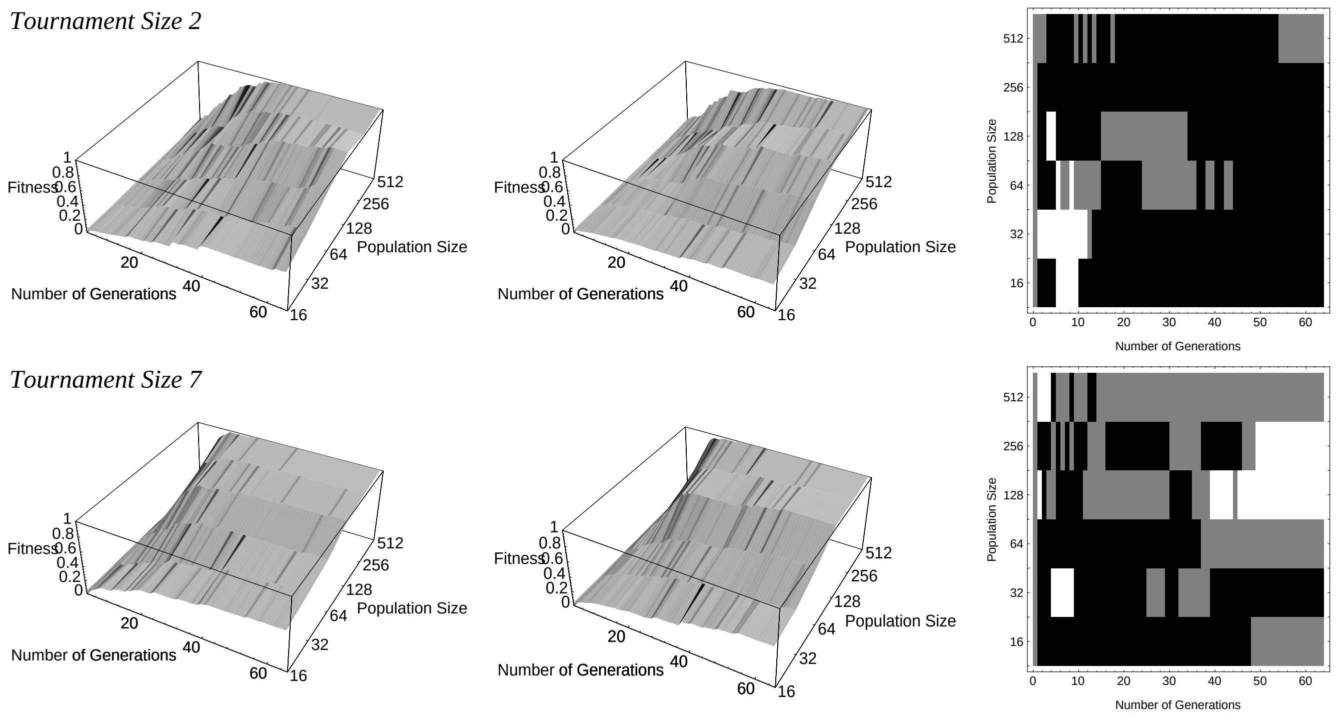


Figure 2. Comparison of crossover and mutation for the Lawnmower problem. Comparison graphs are black where crossover wins, white where mutation wins, and gray where the difference is insignificant.

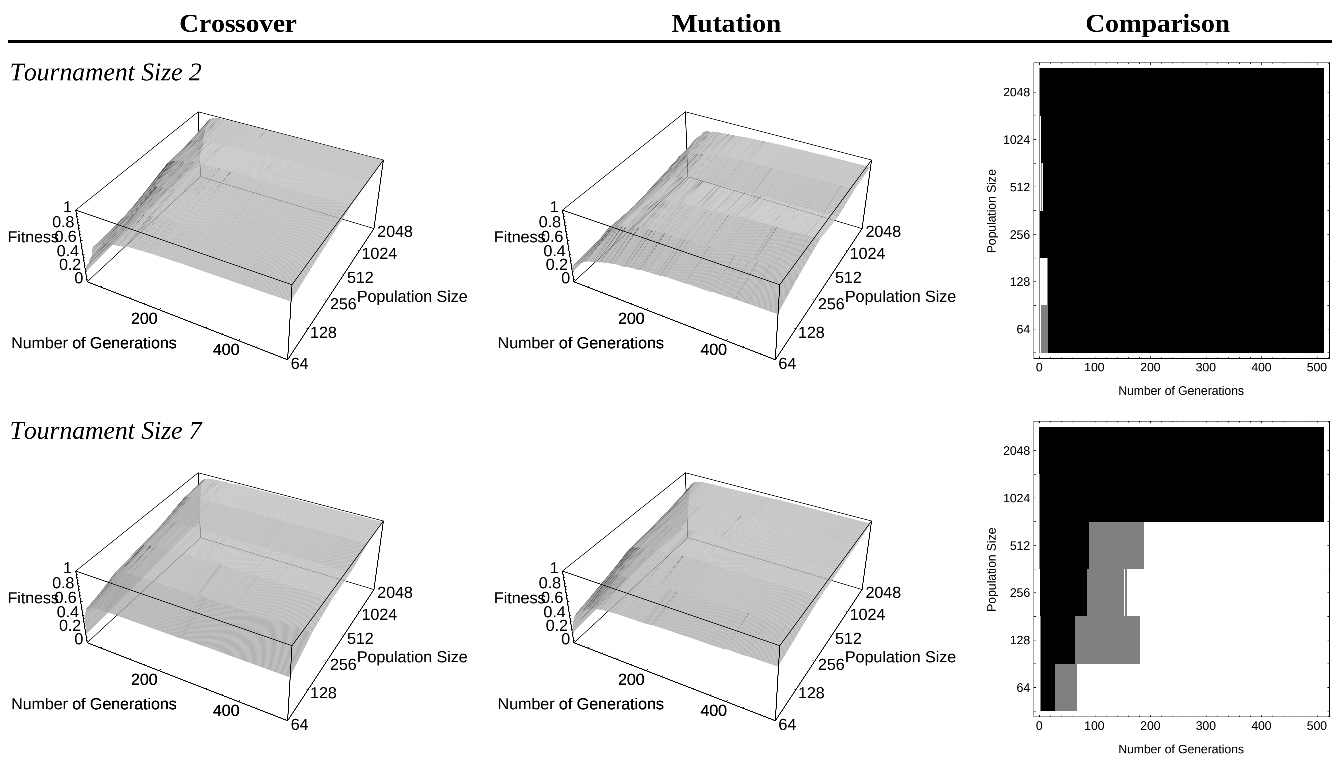


Figure 3. Comparison of crossover and mutation for the Regression problem. Comparison graphs are black where crossover wins, white where mutation wins, and gray where the difference is insignificant.

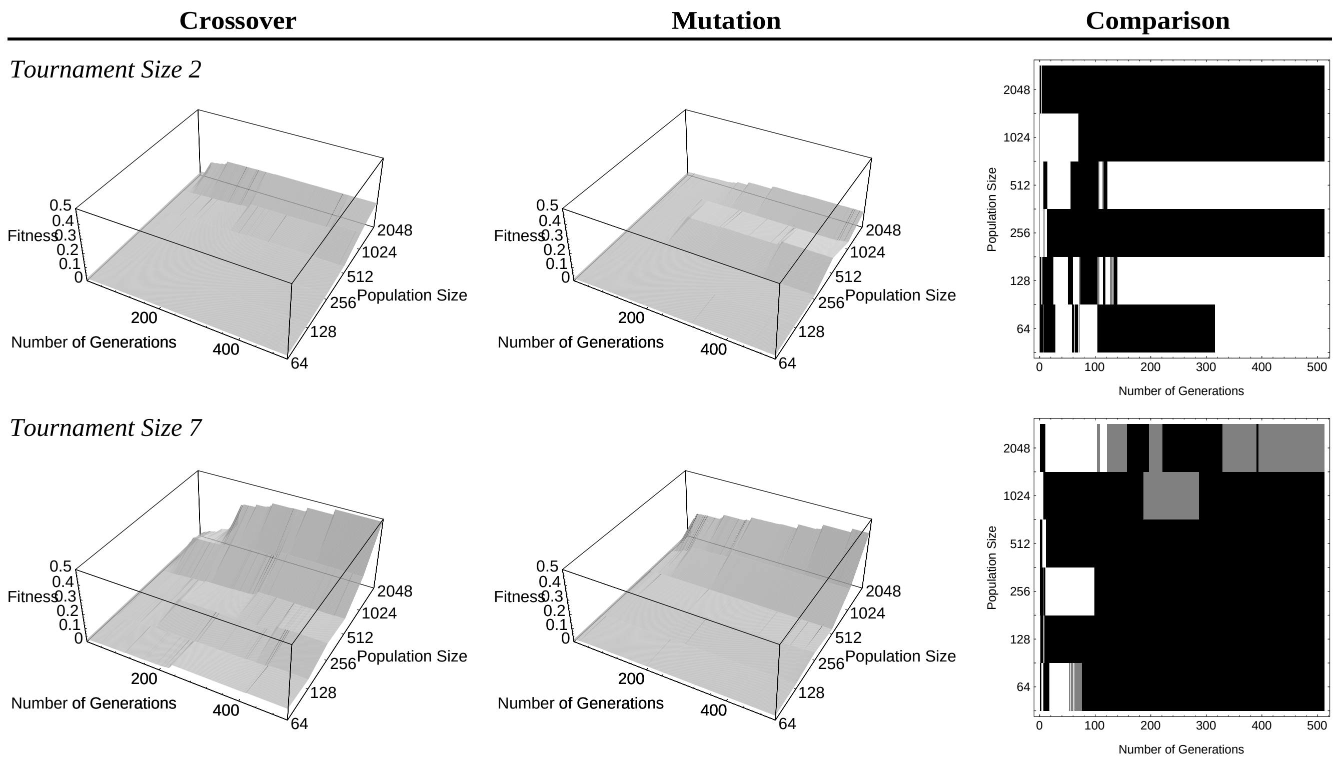


Figure 4. Comparison of crossover and mutation for the Artificial Ant problem. Comparison graphs are black where crossover wins, white where mutation wins, and gray where the difference is insignificant.