Code Growth in Genetic Programming *

Terence Soule
Laboratory for Applied Logic
Computer Science Dept.
University of Idaho
Moscow, Idaho 83844-1010
tsoule@cs.uidaho.edu

James A. Foster
Laboratory for Applied Logic
Computer Science Dept.
University of Idaho
Moscow, Idaho 83844-1010
foster@cs.uidaho.edu

John Dickinson
Computer Science Dept.
University of Idaho
Moscow, ID 83844-1010
johnd@cs.uidaho.edu

Abstract

In this paper we examine how quickly the programs generated using genetic programming grow in size. We found that without a constraint mechanism the programs will grow indefinitely regardless of whether or not the growth acts to improve the programs' solutions. This growth is dominated by non-functional code. If the non-functional code is removed the growth is dominated by functional, but non-executed code. Two methods of controlling the growth: removing non-functional code, and selective pressure applied by penalizing longer programs, are compared. Only the later method appears to be effective in bounding the programs' size.

1 Introduction

Considerable study has gone into how close programs generated using genetic programming (GP) come to the optimal solution: how effective they are [Koza, 1992, Koza, 1994]. The experiments in this paper study how efficiently genetic programming solves problems: how much code is generated compared to the amount of code actually needed. In addition, we measure how much of the generated code is actually non-functional.

*This work supported by funding from the URO FY96 Seed Grant program, and through funding received from the NSF-Idaho EPSCoR project under NSF Cooperative Agreement number OSR-9350539.

One of the interesting features of the programs generated using GP is that they almost invariably grow in size and incorporate large amounts of non-functional code which has no net effect, but does increase the programs' lengths. Unfortunately, there are several reasons for generally preferring shorter programs. Shorter programs typically require less time and less space to run. This is particularly important during the GP process which may need to store and evaluate populations of hundreds or thousands of programs. In addition, shorter programs tend to show better generalization performance than do longer programs. This means that efforts to limit code size will be important in all but the simplest GP applications. Reducing the amount of non-functional code is an obvious place to start.

However, several researchers have hypothesized that the growth of code during GP and the creation of non-functional code are both important contributors to the production of effective programs. Iba et al. have suggested that the creation of large programs is a necessary part of the exploration of the solution space [Iba et al., 1994]. Nordin and Banzhaf have argued that non-functional code is also important in shielding useful code from the harmful effects of crossover [Nordin and Banzhaf, 1995]. They attempted to use external pressures to reduce the amount of non-functional information in variable length genetic algorithm individuals. Although the amount of non-functional information was reduced almost to zero the system failed to find the optimal solution, unlike in the case without external pressure.

In contrast Zhang and Muhlenbein used an implementation of Occam's Razor to limit code size in a successful GP to design neural nets [Zhang and Muhlenbein, 1995]. Their model attempted to balance parsimony and performance. This resulted in populations which cycled between growth and reduction and produced reasonably ef-

fective individuals.

We have used selective pressure to control code size in a relatively successful genetic program for approximating the maximum clique problem [Soule et al., 1996].

Overall, the relative importance of code growth and non-functional code is still unclear. Although there are definite benefits to shorter, more parsimonious programs some studies suggest that code growth, particularly of non-functional code, is important to the evolution of effective programs. This makes attempts to restrict code growth and improve efficiency difficult.

Looking at biological evolutionary processes is also not very enlightening. In humans roughly $80 - 90\%$ of DNA does not code for functional proteins, although some of this DNA does have a structural function. Similar percentages hold for most other higher organisms. On the other hand, prokaryotes have almost no non-functional genetic material. It has been hypothesized that organisms which compete via rapid cell division experience strong selective pressure for streamlined genomes – an efficiency consideration which may be relevant for GP.

Comparing code growth to performance improvements will help indicate how important the growth is in producing effective solutions. The ability to use selective pressure to control code growth and non-functional code without interfering with the production of effective solutions places a limit on the importance of these factors to the GP process. It also leads to more efficient and generalizable solutions.

We examined the rate of code growth in a simple GP, separating the code into functional and non-functional categories. Then we measured the effect of removing the non-functional code or of applying selective pressure. We found that the size of the genetically generated programs grows indefinitely, regardless of whether the additional code acts to improve the programs' fitness. Additionally, most of this growth is caused by increases in the amount of non-functional code. Despite the prevalence of non-functional code, removing this code at every generation does not halt the programs' growth. Instead the programs generate code which, while functional, is never actually executed. These results imply that much of the programs' growth is not caused by pressure to improve the solution, but, instead, is an innate part of the genetic programming process. However, selective pressure was found to dramatically reduce the overall growth and the percentage of non-functional code while still producing effective programs. This implies that most of the growth seen in unrestrained GP and most of the non-functional code is not necessary for the evolution of effective programs.

2 The Experiment

2.1 The Test Arena

Our test case was a GP for a robot guidance problem. The task is to generate a program which will guide a robot through an obstacle-filled room. The room, a simple maze, is shown in Figure 1. This problem was chosen to make automated detection of useless code relatively simple. Branches of statements which fail to change the robots position or direction are clearly useless.

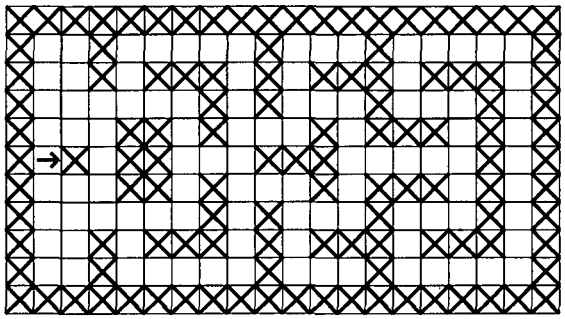

Figure 1: The Room

The initial position and direction of the robot is shown by the arrow at the left of the figure. The fitness of a program is based on the horizontal distance the robot has traveled when the program terminates. Hence, the fitness varies from one to eighteen (the length of the room). No penalties are assessed for running into walls (although of course it does not advance the robot) or for the number of steps taken. To avoid infinite loops the programs are only allowed 3000 statement executions. This number is more than sufficient to cross the room, but should avoid incidentally penalizing programs which might cross the room using a very long path. Of course, some programs will terminate in less than 3000 steps.

We developed a simple, interpreted language for the genetic program (which was written in C++). The language has five terminal statements (which have no arguments and no return values): `left`, `right`, `forward`, `back`, and `no_op`. These statements change the robot's position or direction in the obvious manner. The `no_op` statement has no effect. It was included so that branches of non-functional code could be removed and replaced by a `no_op` without interfering with the syntax of the language, which requires binary branching.

The language also contains three control statements which take two other statements as arguments: **progn**, **if**, and **while**. The **progn** statement simply executes its first argument followed by its second argument. Both the **if** and **while** statements take a predicate, either **wall_ahead** or **no_wall_ahead**. The **if** statement executes its first argument if the predicate is true and its second argument otherwise. As long as its predicate is true the **while** statement executes its first argument and then its second argument. This language produces programs which have the structure of a binary tree. This means that the crossover operation is relatively simple to accomplish through the exchange of branches.

For example, a typical program is

```
progn(left,
while(no_wall_ahead,forward,right))
```

The corresponding program tree and the path along which the program would move the robot are shown in Figure 2. The robot ends at location 2,4 facing the barrier to the South. The program itself would be awarded a fitness of two.

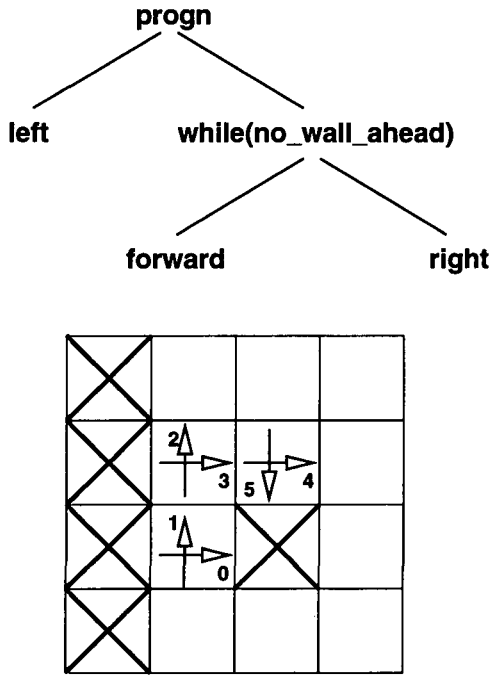

Figure 2: A Sample Program Tree and Path

For each trial the genetic program was run for fifty generations with a population of 500 individual programs. The initial population was generated randomly, but each program always began with a control statement. Thus, the initial programs consisted of at least three statements. The average size of the programs in the initial population was approximately ten statements.

The crossover rate was 0.6667 and the mutation rate was 0.001. Mutation was limited such that statements could only mutate into other statements of the same type: terminals to terminals, controls to controls, and predicates to predicates. At each generation a new population was chosen from the previous one using the stochastic remainder technique which is based on the the fitness ranking of the individuals.

Program fitness was determined by the number of columns between the robot's position at the start and termination of the program. More exacting fitness functions are possible. For example it would be reasonable to consider the number of steps the program used in determining its fitness. However, such fitness functions are likely to exert an additional influence on the code size possibly obscuring the relationships between code growth and performance that we were measuring.

2.2 The Test Cases

We used four test cases: 1) a simple genetic program, 2) a genetic program in which the test programs were reduced at each generation by editing out some of the non-functional code, 3) a genetic program in which only the initial, random population had non-functional code removed, and 4) a genetic program in which the fitness function was modified to penalize longer programs. The use of a penalizing function applies selective pressure in favor of shorter programs.

The non-functional code fell into two basic categories: code which did nothing and code which could never be executed. The first category includes any of the control statements with **no_op**'s for both arguments, **progn**'s with a **no_op** for one of the arguments, and **left-right** combinations. The second category is a result of nested conditionals. For example, consider the code:

```
if(wall_ahead,while(no_wall_ahead,X,Y),Z)
```

where X,Y, and Z are any additional statements, or lists of statements. The statements denoted by X and Y will never be executed. The **while** is only reached if there is a wall ahead in which case the **while**'s conditional is false. Hence, the **while** statement and its attendant branches can all be replaced by a single **no_op** without affecting the program's

performance. The `if` statement must have a statement in its first branch making the `no_op` necessary. Thus, the edited program would become

```
if(wall_ahead,no_op,Z)
```

It is also possible for code to be functional, but to never be executed. For example, the statement `Z` in the above example will never be executed if the robot is always facing a wall when the `if` statement is executed. However, in such a case the non-execution of the statement `Z` is dependent upon the room and is not an inherent feature of the code, so the statement is not considered non-functional and is not removed.

Although these tests remove much of the non-functional code they do not guarantee that it will all be removed. The detection of all non-functional code is an unsolvable problem, since it is reducible to the program equivalence problem, which is non-recursive [Hartley Rogers, 1967]. Therefore all non-functional code cannot be removed. A more exhaustive search could find and remove more of the non-functional code, but the differences between the edited and unedited test cases in our results are definitive enough that more rigorous editing was not warranted.

Interestingly, in a few cases the removal of non-functional code did change the fitness of the edited program. This could occur if, for example, the non-functional code was also an infinite loop. By removing the loop, sections of code which followed the loop were executed where previously they had not been. Similarly, because of the 3000 step maximum, removing an early section of non-functional code might allow a later section of code to be executed. Overall, changes in fitness due to editing were very rare and lowered the fitness roughly as often as they improved it. Hence, the effects were negligible.

The fitness function used to penalize longer programs subtracted one point of fitness for every fifteen statements beyond the first fifty. We felt that the limit should be close to the length of the shortest program which could solve the task. Otherwise there is a significant risk that programs will evolve primarily for size rather than ability. Preliminary trials bore this out. When the cutoff was set too low, program performance suffered. The programs worked to avoid the penalties of a longer program rather than to produce effective solutions. Sample trials found a program of fifty-one statements which successfully crossed the room, so we rounded the the cut–off value to fifty. Further work has shown that shorter programs can successfully complete the task.

3 The Results

All of the data were obtained by averaging the results of fifty separate trials. Figure 3 shows the program length of the best programs of each generation for the four test cases: the standard genetic program (control), the genetic program with individuals edited at every generation (edited programs), the genetic program with an edited initial population (edited initially), and the genetic program with individuals subjected to selective pressure against length (selective pressure). Figure 4 shows the average program lengths averaged across all fifty trials.

For the earlier generations (roughly the first twenty) the length of the best programs are generally greater than that of the average program. However, after that the length of the best and average programs are very similar. This suggests that once the programs reach a certain minimum size a longer program is not any more likely to be effective than a shorter program. The one exception occurs when selective pressure is applied. Here the best programs are actually slightly shorter than the average because the shorter programs are not penalized for length.

Comparison of the edited and unedited programs shows that for the initial random programs almost fifty percent of their length is non-functional code. Simply removing this code in the initial generation has a lasting effect on the code size, although it does not appear to effect the basic growth curve.

We found that non-functional code is responsible for most of the code growth in the control case. The amount of functional code grows much more slowly and, unlike the total code size, its growth begins to level off. This suggests that removing the non-functional code at each generation would restrict the overall program growth. However, the edited programs show that this does not happen. Even when the programs are edited at each generation their growth curve is still very dramatic. We found that when the non-functional code is removed the GP responds by producing large programs consisting entirely of potentially functional code. However, most of this code is never executed, since it is stored in branches of the program which are never reached during execution. This non-executed code accounts for most of the edited programs' growth, while the executed code remains relatively small. More significantly, the growth in the amount of executed code appears to level off just as the growth in the functional code did in the unedited case.

In contrast, penalizing longer code through the fitness function appears to be considerably more effective at reducing code length. More importantly,

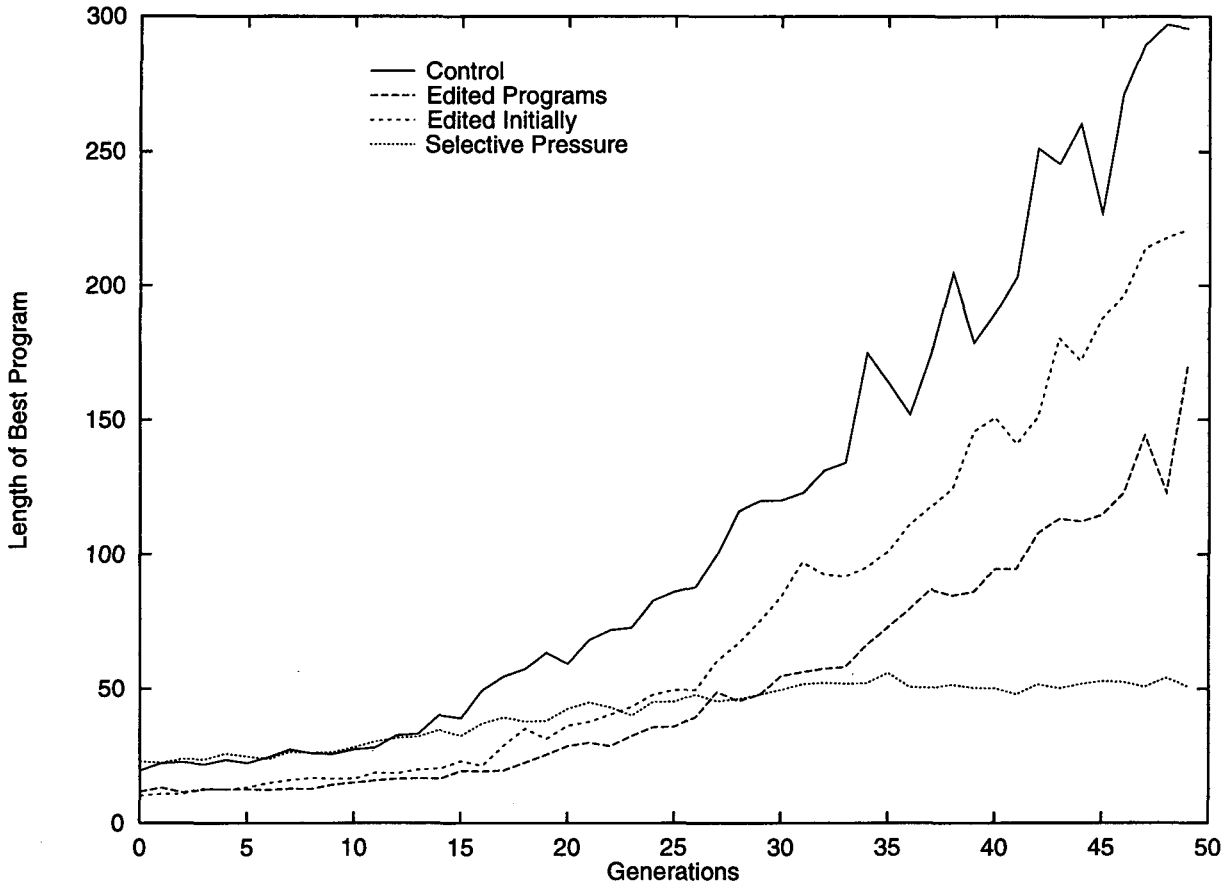

Figure 3: Best Program Lengths

such selective pressure limits the overall amount of growth.

Figure 5 shows the fitness for the best program of each generation for each of the four test cases. The test cases are labeled in the same manner as in the previous figures. All of the cases show roughly the same performance results, including the programs subjected to selective pressure. This strongly indicates that the rapid growth seen in the other test cases is not necessary for the production of effective programs.

Figure 5 also shows that the success of the programs tends to level off towards the later generations. This plateau corresponds to the convergence of the population towards a particular solution and implies that further rapid improvement is unlikely. In some cases the achieved solution is optimal, and in others it is sub-optimal, thus the average fitness over all fifty trials is slightly lower than the the maximum possible (eighteen). Figure 6 shows the average fitness for all of the programs of each generation for the four test cases. The actual fitness values are shown, so the programs subjected to selective pressure are slightly more successful in crossing the room than the data indicate. They have lost points due to their length rather than to poor performance. The data in Figure 4 show that towards the later generations (beginning around generation thirty-five) the average program length with selective pressure was between fifty and sixty. Thus, the fitnesses in Figure 6 are approximately one point less than the distance traveled.

Figure 6 also shows a leveling of the fitnesses towards the end of the run. The plateau lags slightly behind that of the best programs (Figure 5) in a fashion typical of genetic programs. In contrast, Figure 3 and Figure 4 show that no similar effect appears in the code size for the first three test cases. Even though the genetic program is no longer improving significantly (indeed in many of the trials an optimal solution has been found and no further improvement is possible) it continues to make larger and larger programs.

We hypothesize that this phenomenon is caused by crossover. Consider two identical and highly successful programs (program 1 and program 2) which

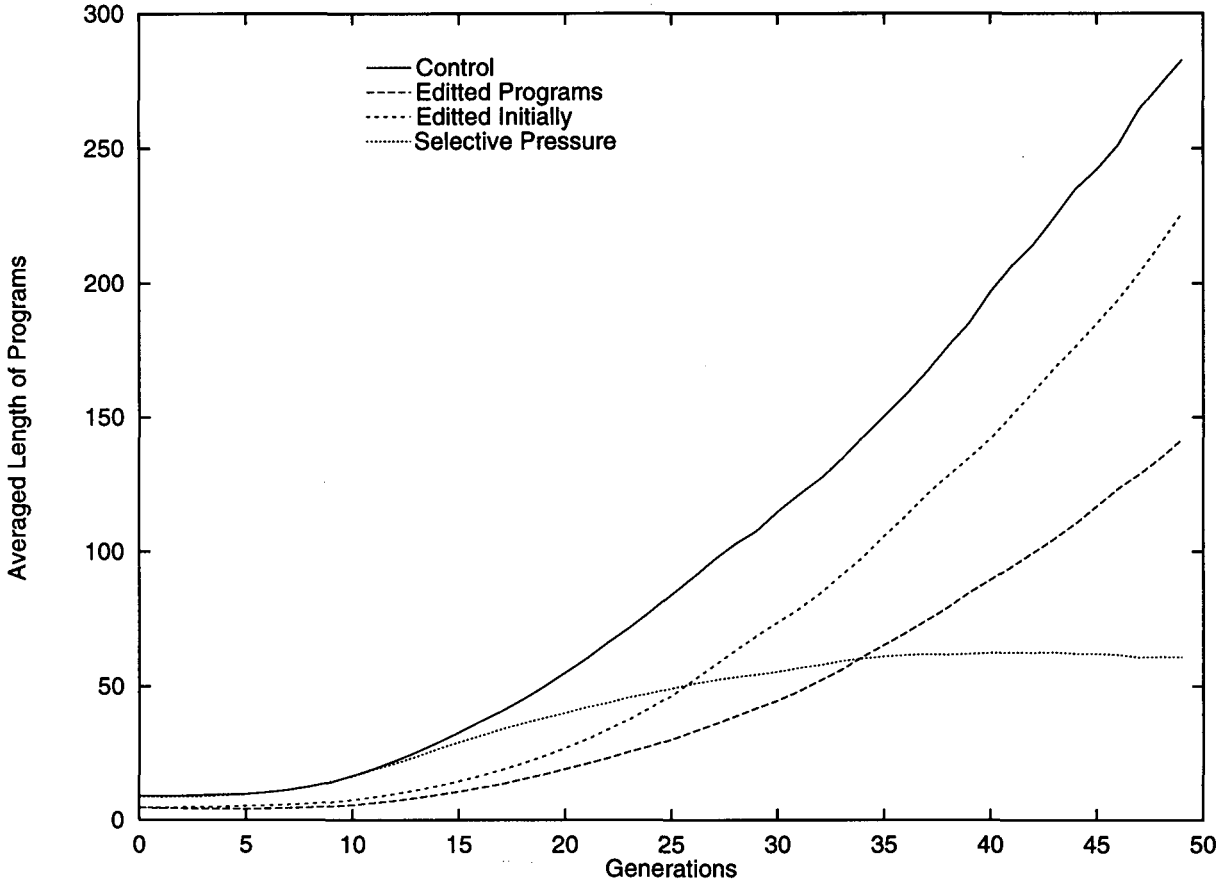

Figure 4: Average Program Lengths

are chosen to be crossed with each other. Program 1 has a relatively small branch removed which is replaced by a much larger branch from program 2. The opposite occurs in program 2; it loses the large branch and gains a smaller branch. It seems likely that after the crossover, program 1, which lost less of its original code and gained more of program 2's code, is going to have a better chance of remaining a fairly fit individual. Whereas program 2, which lost more code and only gained a little to replace it, is more likely to lose fitness. Hence, there is a slight, but distinct, statistical advantage for the longer of two programs created by crossover. This advantage would act to keep the code size growing even after an optimal program had been found, as is occurring in these trials.

This point deserves emphasizing. Even though the programs' performance has begun to level off, they are growing in size just as rapidly as ever. Removing the non-functional code is not sufficient to halt this growth. Only the use of a penalty function and selective pressures are effective, and their success in limiting code size is extremely dramatic.

Given that the penalty function is so successful, it is reasonable to consider what portion of the non-functional code it is removing. Figure 7 shows the average program lengths at each generation for the programs subjected to selective pressure and the average program lengths after these programs have been edited. Clearly, the programs still contain a considerable amount of non-functional code. However, the selective pressure is sufficient to keep the amount of this code from growing indefinitely as in the previous cases.

While we have not considered programs which are limited to less than some fixed size, we feel that some predictions may be made concerning this case. First, to avoid the possibility that even programs of the maximum size cannot complete the task it is necessary to fix the limit at a large value. Given the rate of program growth which we observed, and the fact that most of this growth is either non-functional or non-executed, it is reasonable to expect that the programs will grow rapidly towards the size limit. Additionally, most of this growth will consist of non-functional code.

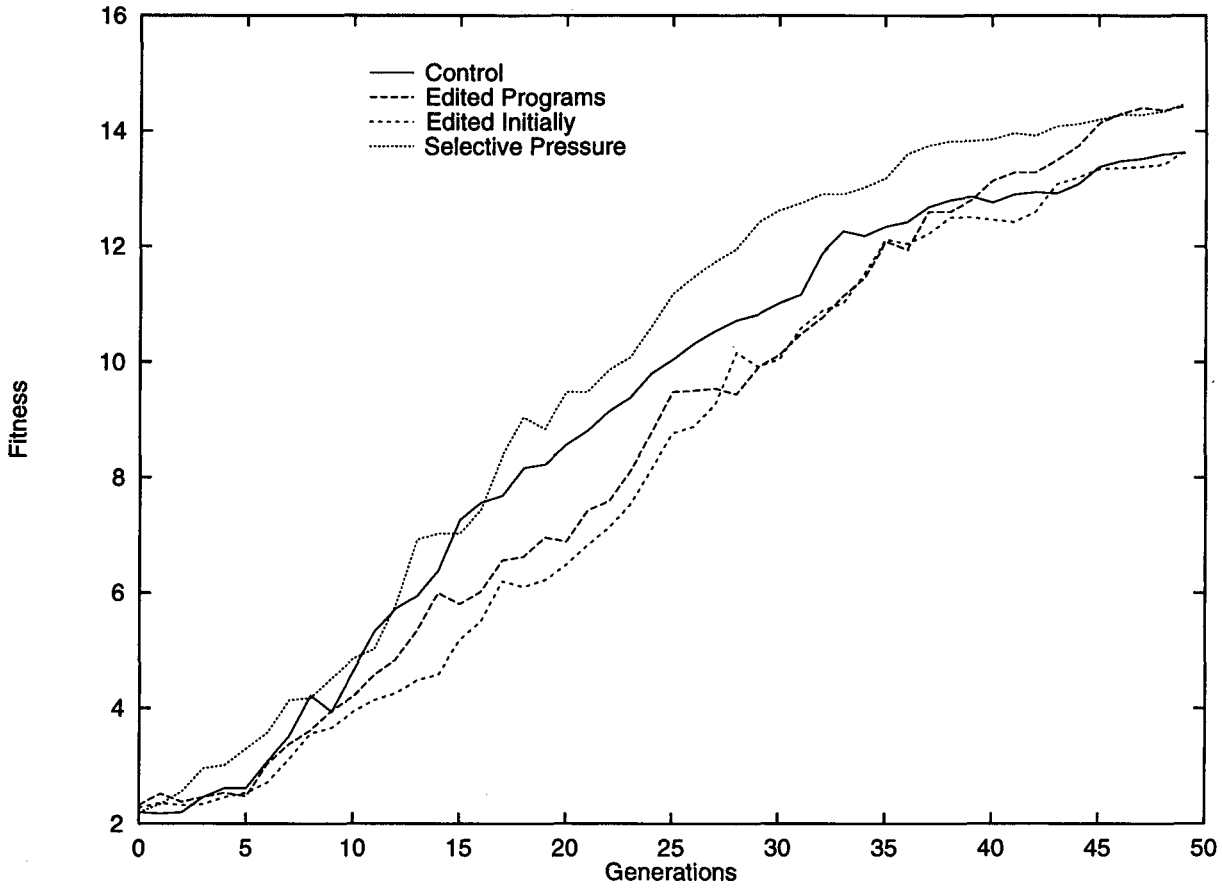

Figure 5: Best Program Fitness

The bound on program size is usually defined as a maximum depth. This restricts the program's size, but also artificially influences its shape. For programs to incorporate the maximum number of functions they must be relatively symmetrical. An additional problem stems from the fact that the enforcement of the size limit normally occurs during crossover. If a program created during crossover exceeds the accepted limit it is discarded and one of its parents is used in the next generation instead. This requires additional computation to detect oversized offspring and, perhaps more importantly, also artificially influences the programs' development. Effectively, there is a decline in the crossover rate as the average program size grows and more products of crossover are discarded.

Using a fitness function which penalizes larger programs avoids most of these problems. It is true that programs appear to rapidly approach the size at which penalties are introduced. However, because this limit is not completely restrictive, it can be set much lower than otherwise. This means that the usual program size will be much lower without running the risk of making the task unsolvable.

4 Conclusions

Code growth in GP is an issue which certainly deserves attention. In these experiments code growth was extremely rapid and gave every indication of continuing, without corresponding increases in fitness. We hypothesize that this growth is due to a slight preference for the larger of the two programs created during crossover. This would cause an overall increase in the program sizes. However, this doesn't appear to be an important factor in evolving effective programs. Incorporating a balancing preference for shorter programs should restrict this growth without harming the overall results.

Most of the code growth is attributable to increases in the amount of non-functional code. However, removing this code does not halt the growth– instead functional, but non-executed, code is substituted for non-executable code. Removing non-functional code does not appear to harm the evolution of effective programs and does lead to program

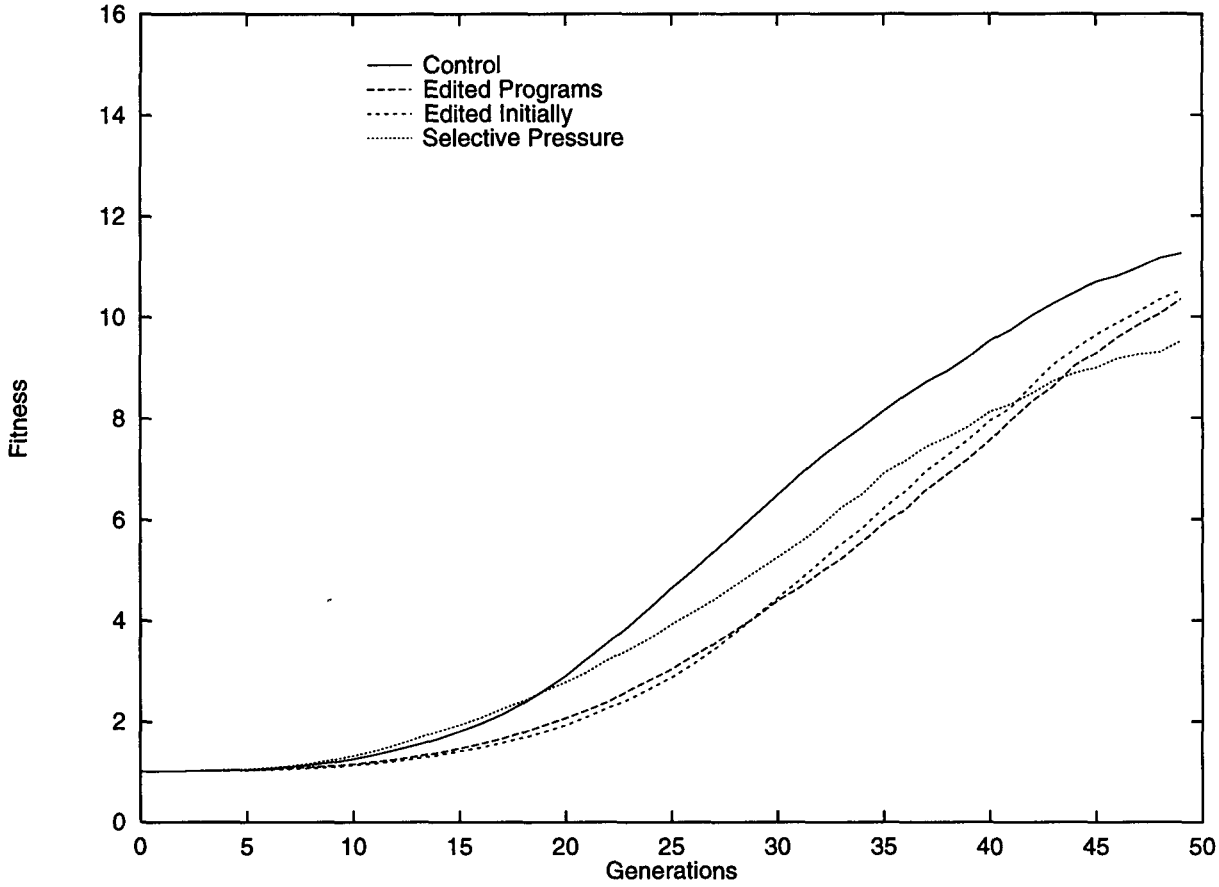

Figure 6: Average Program Fitnesses

which are considerably shorter than when editing is not used.

Using a fitness function which penalizes large programs generates selective pressure which does act to limit program growth. Furthermore, in our experiments selective pressure does not hurt the production of effective programs. Thus, it appears that selective pressure can be used to evolve programs which are both effective and efficient. It is evident that all of the code growth and non-functional code seen in our control and edited cases, as well as many other GP applications, is not necessary to the evolution of effective programs.

The one difficulty in using a penalty function is that it may require estimating the minimum size of programs which can solve the problem. However, the growth of unrestricted programs is so rapid and the effect of the penalty function is so dramatic that even an extremely liberal guess at the minimal program size should produce beneficial results. The effects of underestimating the required program size, and thereby penalizing programs which are too small to optimally solve the problem still needs to be studied, as does effectiveness of other types of penalty functions.

Other factors being equal, shorter code has the inherent benefits of requiring less resources and generalizing better than longer code. However, it is also possible that shorter code is also more amenable to the GP process. If, as Nordin and Banzhaf suggest, non-functional code shields functional code from the harmful effects of crossover, it also decreases the chances of helpful crossovers occurring. Thus, too much code growth would lead to a stagnation of the GP process.

References

[Hartley Rogers, 1967] Hartley Rogers, J. (1967). *Theory of Recursive Functions and Effective Computability.* McGraw–Hill.

[Iba et al., 1994] Iba, H., de Garis, H., and Sato, T. (1994). Genetic programming using a minimum description length principle. In Kenneth E. Kinn-

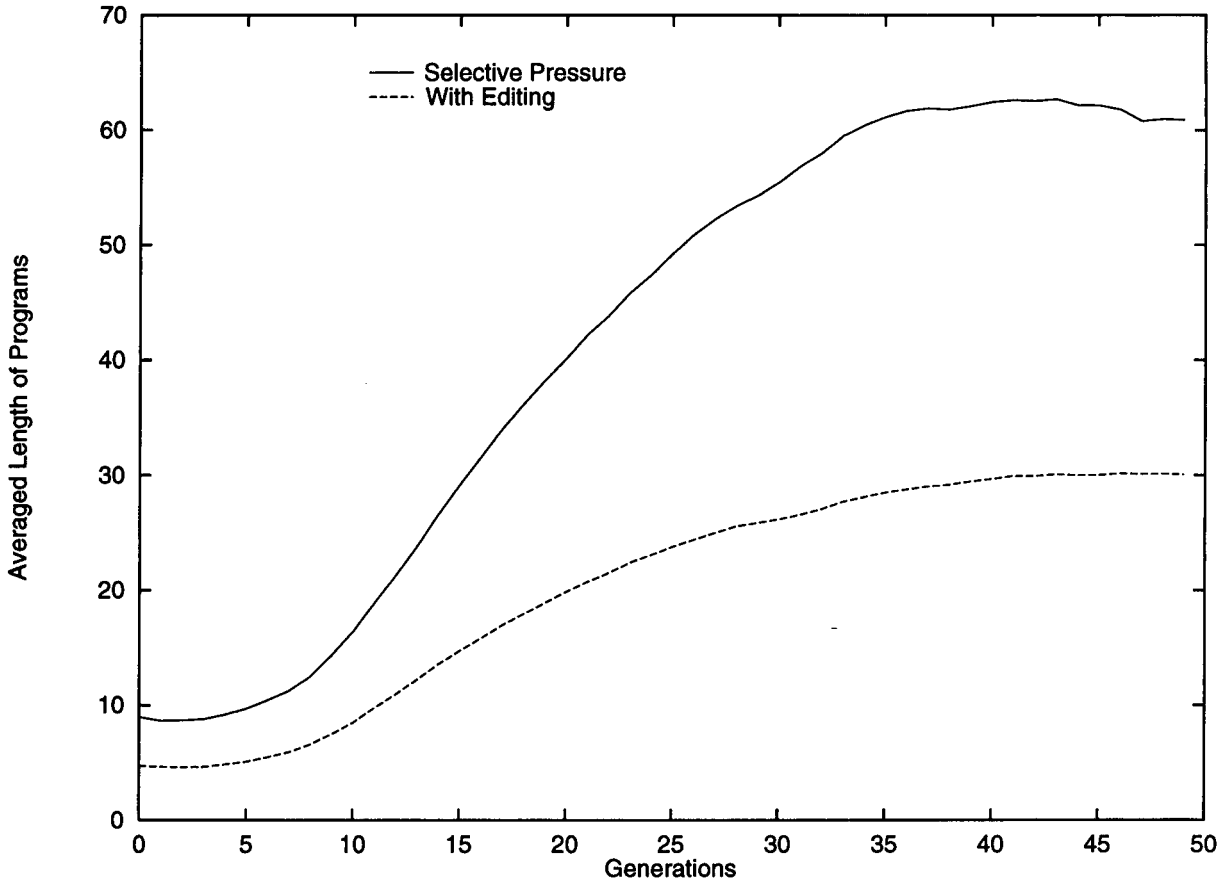

Figure 7: Non-functional Code in Penalized Programs

ear, J., editor, *Advances in Genetic Programming*, pages 265–284. Cambridge, MA: The MIT Press.

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

[Koza, 1994] Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: The MIT Press.

[Nordin and Banzhaf, 1995] Nordin, P. and Banzhaf, W. (1995). Complexity compression and evolution. In Eshelman, L. J., editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 310–317. San Francisco, CA: Morgan Kaufmann.

[Soule et al., 1996] Soule, T., Foster, J. A., and Dickinson, J. (1996). Using genetic programming to approximate maximum clique. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. R., editors, *Genetic Programming 1996 (this conference).*

[Zhang and Muhlenbein, 1995] Zhang, B. and Muhlenbein, H. (1995). Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38.