Genetic Programming and Evolvable Machines, 5, 271–290, 2004

Problem Difficulty and Code Growth in Genetic Programming

STEVEN GUSTAFSON smg@cs.nott.ac.uk
School of Computer Science & IT, University of Nottingham, NG81BB, UK

ANIKÓ EKÁRT ekart@sztaki.hu
Computer and Automation Research Institute, Hungarian Academy of Sciences, 1518 Budapest, P.O.B. 63, Hungary

EDMUND BURKE ekb@cs.nott.ac.uk
GRAHAM KENDALL gxk@cs.nott.ac.uk
School of Computer Science & IT, University of Nottingham, NG81BB, UK

Submitted March 4, 2003; Revised August 7, 2003

Communicated by: Riccardo Poli

Abstract. This paper investigates the relationship between code growth and problem difficulty in genetic programming. The symbolic regression problem domain is used to investigate this relationship using two different types of increased instance difficulty. Results are supported by a simplified model of genetic programming and show that increased difficulty induces higher selection pressure and less genetic diversity, which both contribute toward an increased rate of code growth.

Keywords: genetic programming, population diversity, code growth, problem difficulty

1. Introduction

Two challenging problems in genetic programming are the application of genetic programming on increasingly difficult problem instances and the increase of solution size which is not correlated with improvement. Daida et al. [3] recently demonstrated that when genetic programming has difficulty solving harder problem instances, solutions are generally larger, take longer to find, and are less numerous in the population. Of these traits, the increase in size without corresponding fitness improvement is the most worrying. The continued growth of solutions for difficult problems will become a limiting factor of the applicability of the algorithm; computational resources will be exhausted and the algorithm will halt before reaching a good solution. The existing theories of code growth [16, 19, 33], or bloat, provide the mechanical basis for understanding why programs in a variable-length representation tend to grow in size, but do not predict the rate of growth or explain why optimal solutions can vary in size by a large degree. Additionally, the many methods used to limit code growth, or to remove non-functional code from solutions, often lead to worse performance [5, 20, 32, 35].

What induces the algorithm to produce much larger solutions when smaller ones do exist? Is it due to the stochasticity of genetic programming, the complexity required by the problem instances, or could there be an additional factor of increased difficulty that encourages a higher rate of code growth? This paper investigates increased problem difficulty and code growth with the goal of understanding if an underlying relationship exists between the two. If increasing the difficulty of the algorithm to find a solution induces a higher rate of code growth, then we might also understand why the size of solutions can vary within instances.

1.1. Our contributions

In this paper we use two symbolic regression problems that are differently scaled in difficulty to observe the behaviour of fitness, code growth and diversity. Maintaining diversity is commonly cited as a way to prevent the algorithm from converging to local optima. If growth is related to instance difficulty, it will be likely that the dynamics of the population, viewed with measures of diversity, will aid in understanding the phenomenon.

We use symbolic regression problems in this study for two main reasons: First, there is a large body of existing theoretical studies using regression problems to understand genetic programming. Secondly, it is easier to reason about increasing the difficulty of fitting mathematical functions than it might be for increasing the difficulty of other types of problems.

A causal relationship between problem difficulty and diversity that leads to increased code growth is found. We initially reinforce these results by examining a large body of related literature, and then we also verify results with a simplified model of genetic programming. The results indicate that increased difficulty leads to higher selection pressure and less diverse populations, both of which contribute to an increased rate of code growth. We then make recommendations on how to reduce the rate of growth in the context of problem difficulty.

1.2. Previous work

What makes a problem difficult with respect to genetic programming? Daida et al. [3] investigated a tunable problem, the binomial-3 function with varying ephemeral random constant (ERC) ranges, to show that difficulty can be increased without changing the combinatorial search space. In this case, genetic programming difficulty increases with the increased range of ERC values. The authors suggest that the conflict between content and context is largely responsible for increased difficulty. O'Reilly and Goldberg [9, 25] use constructed problems to also highlight the content and context dependencies in genetic programming solutions. They also investigated how partial solutions contribute differently toward fitness and how their definition makes solving the problem more difficult [26].

Different problems are likely to pose their own nature of difficulty to genetic programming. For example, in the class of boolean problems (parity, multiplexer), the processing of all boolean variables is generally required to solve all fitness cases. Therefore, the ability for each candidate program to acquire all boolean variables will effect difficulty.

As this study is primarily focused on understanding the relationship between difficulty and code growth, and whether difficulty somehow influences the rate of code growth, we will not be investigating in detail what makes a problem difficult. Instead, we will use two types of regression problems for which it is fairly straightforward to understand how difficulty

might arise. *The longer time taken for finding solutions, the increased size of solutions, and the appearance of fewer solutions* are all characteristics of an increase in difficulty. This behaviour is demonstrated very well by the binomial-3 problem [3].

Several recent studies of the causes of code growth exist. Luke [19] presents a theory based on the depth of modification points and Soule and Heckendorn [33] assess the viability of three hypotheses, protective, drift and the removal bias, respectively. Both studies provide very thorough discussions of their theories, and we present only a brief summary of the key elements for later analysis.

There are two major contributing factors to code growth in genetic programming which we would like to emphasise here:

1. The smaller the removed portion of a tree, the less likely that the fitness of the resulting offspring will be worse than the parent's (according to the *removal bias* [11, 33]).
2. Child survivability is linked to deeper nodes selected for modification in their parents (according to the *depth-based theory of code growth* [19]). This gives a bias toward larger parents which will tend to produce larger children.

These two points are overlapping, and are also somewhat consistent with other theories of code growth, specifically Langdon and Poli's *fitness causes bloat* theory [13, 15, 17], which suggests that the space of larger trees contains more better fit programs. As fitness increases or stays the same with tree size growth, the selection of crossover points nearer to the leaves increases, the size of the removed portion decreases and thus the modification points are deeper. There is a clear bias toward a constant rate of growth during evolution. However, empirical evidence also shows a wide range of code growth rates, which these theories do not predict.

Given that genetic programming is likely to favour larger trees, several methods have been used to attempt to limit code growth. The standard method to deal with code growth is to place a limit on the size of individuals [12, 20, 28]. This measure has been shown to be difficult to beat with respect to overall algorithm performance. However, ideal sizes for particular problems are generally unknown, and placing a limit on size can have unforeseen consequences as trees will find it harder to replicate within size limits. Also, when genetic programming is applied on more complex problems it is not clear how to scale the limits of size appropriately.

Parsimony pressure is another common method to control size [1, 10, 20, 21, 29]. This method favours fitter and smaller individuals. The degradation in performance due to parsimony pressure is studied by Soule and Foster [32]. They noted that pressure to reduce size increases the failure of runs, where populations often converge to very small, poorly fit individuals. Several variations of such methods exist which vary the relative importance of size and fitness. Their drawback is that they require a large parameter search for optimality [20]. In [35], an adaptive pressure allowed an initial increase in size before increasing the parsimony pressure. This allowed the method to find better fitness but required problem specific tuning. Other methods add a size objective to the fitness objective, creating a multi-objective problem [4, 8]. Smaller size individuals and similar fitness with smaller population sizes are reported benefits of these methods. Using specific recombination operators and methods which combine like-sized individuals [14] or requiring an improvement in fitness in children [27] are other approaches to reducing code growth. These methods attempt

to reduce the amount of ineffective code. Generally, without a large parameter search or problem specific modifications, these methods are likely to reduce code growth but also worsen fitness.

Given the characteristics of difficulty in genetic programming (code growth, longer time until solutions and fewer solutions) and that, under normal conditions, code growth is generally expected, we look at two ways of increasing problem difficulty in the next section.

2. Regression problems and increased difficulty

Building on the study by Daida et al. [3], we use the binomial-3 problem with varying ERC ranges to increase difficulty. The binomial-3 problem consists of approximating the function $f(x) = (1 + x)^3$ using the functions $\{+, -, p/, \times\}$ and the terminals $\{x, R\}$. R are ERCs in the ranges of $[-1, 1)$ (referred to later as 'unity'), $[-10, 10)$ ('ten'), and $[-100, 100)$ ('hundred'), where larger ranges create increasing difficulty. Several other ranges were used in the original study. The binomial-3 function is shown graphically over the interval $[-1, 0]$ in Figure 1 (top left). There are many solutions to this function which can be represented easily by many different combinations of functions and terminals. The easiest version of this problem would be expected to allow for many close-to-optimal initial solutions and

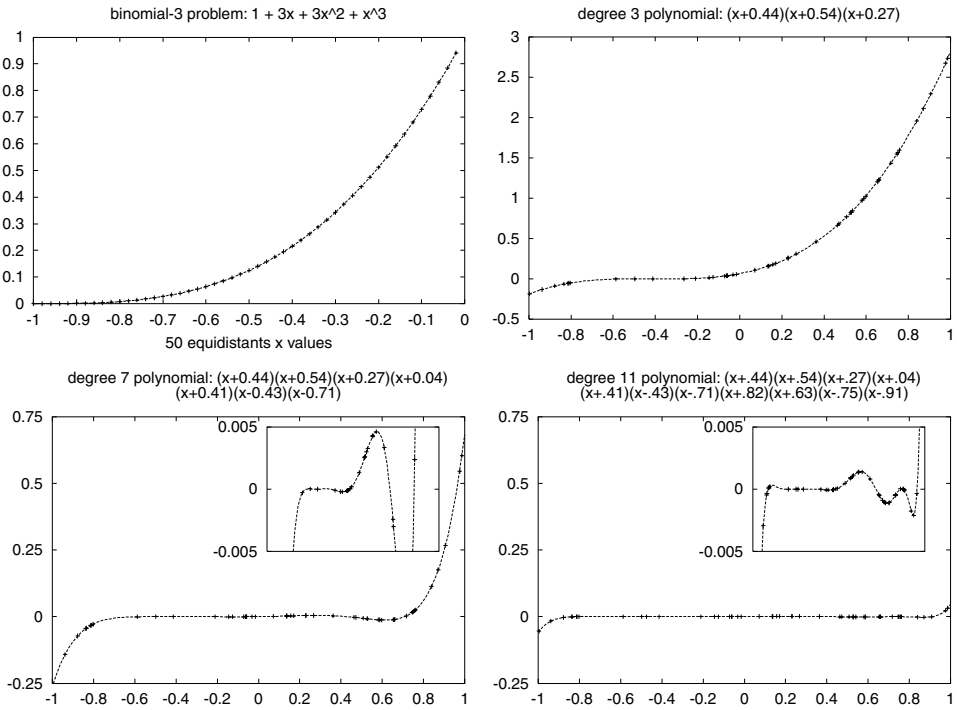

Figure 1. The binomial-3 problem and the generated polynomial functions of degree 3 (upper right), 7 (bottom left) and 11 (bottom right). The inset for degree 7 and 11 shows the same x-range and smaller y-range between $[-0.005{:}0.005]$, where it is relatively easy for genetic programming to find a *close* fit, but difficult to fine-tune the approximation.

more new optimal solutions to be acquired in every generation. Harder instances are likely to cause more difficulty in finding good solutions early and often.

To complement this problem, we use the method of random polynomial generation described in [7]. We generate random polynomials of degree up to 11 with non-zero coefficients and with real roots in the range $[-1, 1]$. As higher degree polynomials have a larger number of roots close to zero, most of the values they take in the $[-1, 1]$ interval are close to zero. For genetic programming, it is easy to find relatively close fit solutions in the form of straight lines close to the X axis. However, it is difficult to improve on these solutions to find a really good approximation, especially if the relatively fit and simple individuals spread across the genetic programming population after a few generations.

The polynomials of increasing degree with non-zero coefficients and real roots in the $[-1, 1]$ interval constitute a class of increasing difficulty problem instances which are suitable for studying the behaviour of genetic programming, as they have the following properties:

- *Minimum Required Depth.* Higher degree polynomials can be approximated less easily as, with our function set, each term of degree d is formed by a sequence of $term_1 \times term_2 \times \cdots term_d$. This, in general, will require larger solutions for a good approximation on a higher degree polynomial. A polynomial with real roots and non-zero coefficients $P(x) = \prod_{i=1}^{d} (x - a_i)$ can be expressed as a tree by using $2d - 1$ functions (i.e., $d - 1$ multiplications and d subtractions) and $2d$ terminals. The minimum depth (thus complete) tree representing this polynomial is of depth $2 + \log(d)$.
- *Maximum Allowed Depth.* It follows from the above requirement that polynomials with higher degree are harder for genetic programming to approximate if allowing the same limited depth for trees, as there is less space for genetic programming to grow introns. In these cases the trees must be more efficient in using more nodes to express the solution.
- *Approximation Ability.* Lastly, by using the Matlab Polyfit routine, we attempt to assign a degree of approximation ability to the polynomials. Polyfit finds the best polynomial of given degree to fit a set of points. Our polynomials of degrees 11, 7 and 3 (shown in Figure 1) were best fitted as polynomials of degrees 11, 8 and 3, respectively, with errors of order 10^{-15}, the approximation error being the highest for the 11 degree polynomial. That is, there were no close approximate polynomials of degree i to generated polynomials of degree j with $i \ll j$. Furthermore, as our generated polynomials are best fit with Polyfit polynomials of similar degree, a successful application of genetic programming on this problem should also find polynomials of similar degree.

The type of increased difficulty posed by the random polynomials and the binomial-3 function is similar. The binomial-3 problem must cope with ERC values spanning an increasing range. Likewise, for the random polynomials, when the degree increases, the fixed ERC range will become less appropriate as the polynomials contain variations of smaller magnitude.

For assessing the differences in behaviour between easy and difficult instances, we use two measures, as detailed below. An entropy based measure is used for indicating the distribution of individuals over fitness values (and subsequently the level of selection pressure) and an edit distance based measure is used for indicating the structural diversity of a population.

2.1. *Population measures: Entropy and edit distance diversity*

To better understand the distribution of fitness values that our selection mechanism is presented with, and the ability of the population to represent solutions for each problem instance, we use a measure of entropy based on fitness. The entropy measure represents the chaos of the system with respect to the distribution of fitness values amongst the population. Rosca [30] introduces the entropy measure for genetic programming that we will use here. We define a fitness class p_k, where there are $1 \ldots k$ unique fitness values in the population, as the proportion of the population having this fitness value. Thus, if there are 500 individuals and 499 have fitness 0.0 and 1 has fitness 1.0, then there are two fitness classes which have the p_k values $p_1 = 499/500$ and $p_2 = 1/500$, respectively. We can then calculate the entropy as:

$$-\sum_k p_k \log p_k.$$

An increase in entropy reflects the system (population's fitness values) moving into a state of more chaos (more different fitness values). This measure also gives us a sense of how the fitness values are distributed over the fitness classes, from a uniform distribution over all fitness classes to the other extreme of the entire population belonging to only one fitness class.

Measuring the genetic diversity of populations is difficult, as there are many aspects of tree shapes and contents that could be measured. In this study we use a measure based on the edit distance between two trees. The edit distance [4] of two trees is computed by overlapping the trees at the root, counting the non-identical overlapping nodes, and normalising the result by dividing it by the smaller tree size. We then compute the average distance of an individual in the population to the current best of run individual. This measure of edit distance is not specific to recombination operators, only to one-point mutation operators that are not used here. This measure is used to understand how structurally similar populations become and to give insight into the reason and the kind of more or less effective search. This edit distance measure is similar to the Levenshtein measure used by O'Reilly [24] and Igel and Chellapilla [11] and the metric which considers depth, by Ekárt and Németh [6], and later used by Burke et al. [2], as it considers single operations on nodes to make two trees equal. Other measures are possible which do consider specific recombination operators but are thought to be too computationally expensive.

3. Experimental investigation

The genetic programming framework described here uses standard parameter settings found in many studies [3, 23, 33]. A population size of 500, a generational algorithm with 101 generations, and standard subtree crossover with internal node selection of 0.9 are used. Tournament selection is used for recombination with a tournament size of 4. Ramped half-n-half initial tree generation with maximum depth of 4 and maximum depth of trees during evolution of 10 are employed. 100 independent random runs are performed for each problem instance. The Evolutionary Computation in Java [18] version 7.0 is used with the Mersenne Twister random number generator [22]. Function sets of $\{+, -, *, p/\}$ and terminal sets

of $\{x, R\}$, where R are ERCs set to a range of $[-1.0, 1.0)$ for the polynomial instances and to the ranges of $[-1, 1)$ (unity), $[-10, 10)$ (ten), and $[-100, 100)$ (hundred) for the binomial-3 problem. The protected division operator is used, which returns 1.0 if zero is found in the denominator. Fitness for both problems is calculated by summing the squared difference for each point along the problem instance's function and the function produced by the individual. All problems used here are minimisation problems with ideal fitness of zero. However, we report the adjusted fitness 1/(1 + raw fitness) (with ideal value of one). The adjusted fitness corresponds to the more natural view of improving fitness by increasing it. For the random polynomial experiments a random seed was selected and used to generate three polynomials of degree 3, 7 and 11. The three polynomials are:

$$\text{degree } 3 = (x + 0.44) \times (x + 0.54) \times (x + 0.27) \tag{1}$$

$$\begin{aligned}\text{degree } 7 = {} & (x + 0.44) \times (x + 0.54) \times (x + 0.27) \\ & \times (x + 0.04) \times (x + 0.41) \times (x - 0.43) \times (x - 0.71)\end{aligned} \tag{2}$$

$$\begin{aligned}\text{degree } 11 = {} & (x + 0.44) \times (x + 0.54) \times (x + 0.27) \\ & \times (x + 0.04) \times (x + 0.41) \times (x - 0.43) \times (x - 0.71) \\ & \times (x + 0.82) \times (x + 0.63) \times (x - 0.75) \times (x - 0.91)\end{aligned} \tag{3}$$

The polynomials are graphically presented in Figure 1.

4. Results

The results are evaluated to first validate the increase of difficulty across instances. We then look at the measures of entropy, edit distance diversity and size and develop a general hypothesis as to what induces faster rates of code growth. The following section looks more closely at the correlation between growth and these measures.

4.1. Establishing difficulty

Daida et al. [3] established the tunable nature of the binomial-3 problem by demonstrating increased difficulty through the reduced frequency which good solutions are found, the longer it took for good solutions to be found and the increased size of good solutions. We reproduce their results and show similar ones for the polynomial problem. In Figure 2, the best fitness of 100 runs for each experiment are placed into bins and plotted in a histogram. The left-most bins in each experiment—corresponding to the number of best solutions found—are expectedly larger for easier instances. The binomial-3 results show a clear trend of less good solutions with increasing ERC ranges. In the case of the polynomial results, this trend is clear for degree 3 and degree 11. Note the large number of poorly fit runs for the degree 11 polynomial (the right-most bins). These 'failed' runs tend to be present in regression problems where the lack of initial fitness improvements causes the run to become stuck in poor local optima. The nature of the degree 11 polynomial amplifies this effect, as we would expect. As shown later, the degree 7 polynomial produces a mixture of results but generally shows the expected behaviour.

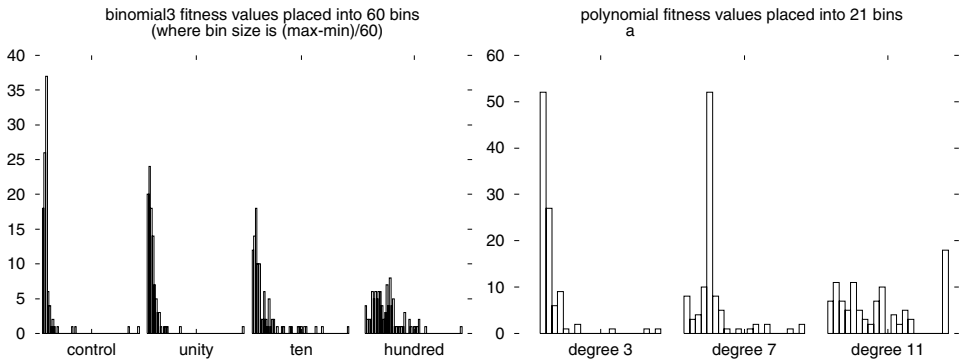

Figure 2. Each problem instance's final population performance values have been discretized and placed into 60 (binomial-3) or 21 (polynomials) bins to show their distribution and relevance for the performance groups. Bins to the left in each instance represent better fitness.

In Figure 3 we reproduce Daida et al.'s results and in Figure 4 we show the results for the generated polynomials. The same trends from [3] apply for both the binomial-3 instances and the polynomials in Figure 4. Increasing the ERC range decreases the best fitness found, increases the size and depth of the solutions when best fitness is found. When the difficulty is increased the best-of-run solutions are found in later generations. Note that while the fitness function remains the same in the binomial-3 problems, the polynomial fitness function does change. For the polynomial experiments, we are most interested in the amount of improvement that the genetic programming can find. The difference in this behaviour can be seen in Figure 6.

Because the higher degree polynomials contain smaller variations, the fitness values across instances are not the same. As the higher degree polynomials start with lower initial fitness, we observe the improvement rate from initial generations to later ones as a sign of performance. The higher the degree, the less improvement is made during the evolutionary process, the bigger and deeper solutions are and the later they are found. We now proceed to a more in-depth analysis of the behaviour of runs with respect to size, entropy and edit distance.

4.2. Binomial-3 problem

For the increasing range of ERCs, the problems (control, unity, ten and hundred) become increasingly difficult for genetic programming to solve. As shown in Figure 5, fitness converges less quickly, fewer good fitness values are found and individuals become larger and deeper for the larger ERC ranges. Additionally, we see that edit distance diversity is slightly higher for the easier instances. This would seem counter-intuitive as one would expect earlier convergence toward similar programs for easier instances. In the bottom graph of Figure 5, we see that the entropy quickly falls and rises initially, then continues to decrease. This would confirm that once good solutions are found, the population loses unique fitness values, more and more solutions have the same fitness value, leading to lower

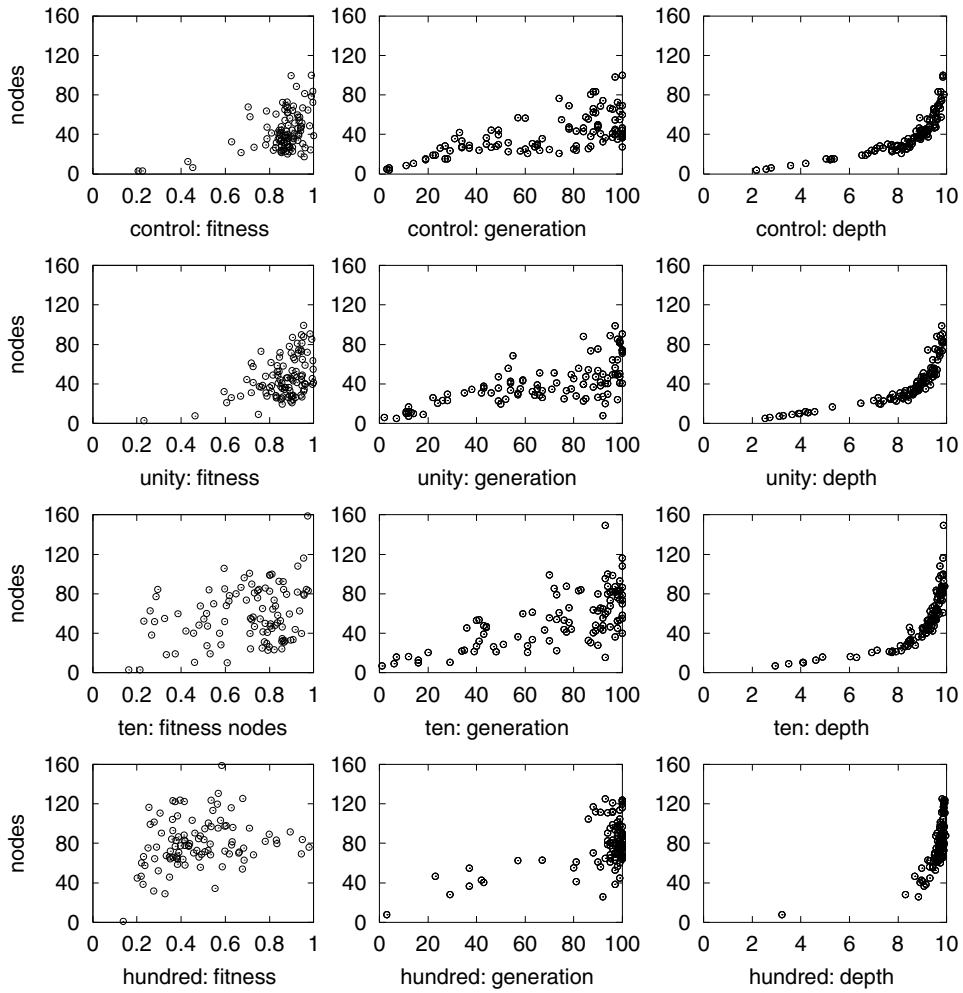

Figure 3. Results of the binomial-3 experiments. The average size of the population containing the best-of-run individual is plotted against best-of-run fitness (left column), the generation in which the best-of-run individual occurred (middle column) and the average depth of the population (right column). Each circle corresponds to one run.

entropy. However, the easier instances are also more likely to be solved by more *different* programs. This explains why diversity is higher in this case. While the populations do converge and lose diversity, the easier instances converge to a more varied selection of fit individuals.

Note that the time when entropy begins to slowly decrease also marks the time when code growth begins to slow down. For the easier instances this occurs just before generation 40, and for the hundred instance, somewhere between generation 60 and 80.

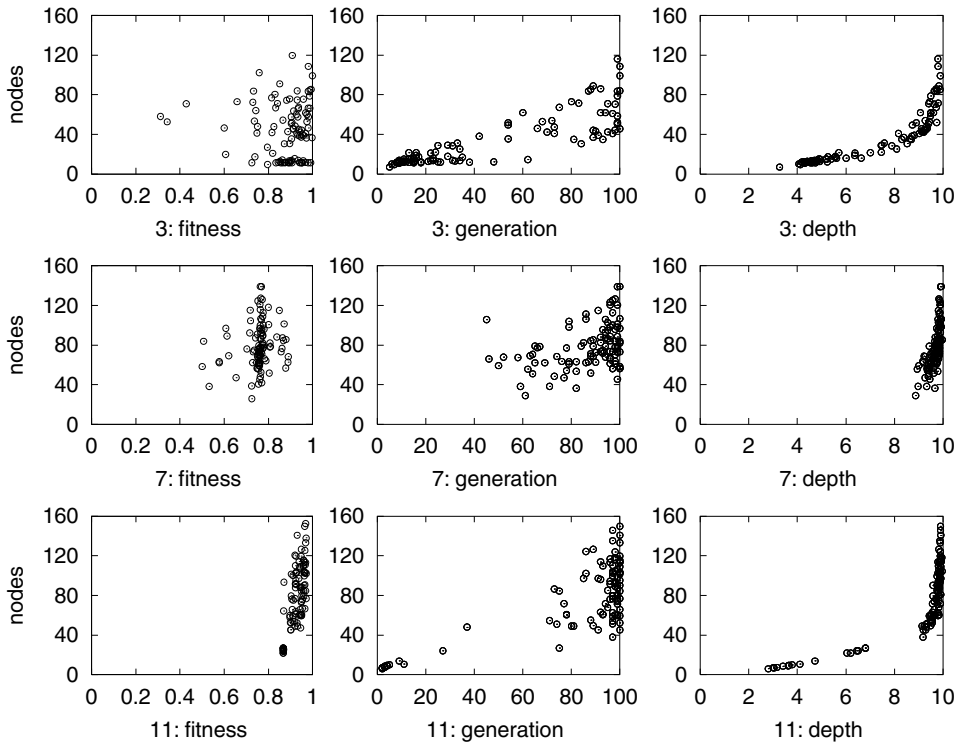

Figure 4. Results of the random polynomial experiments. The average size of the population containing the best-of-run individual is plotted against the best-of-run fitness (left column), the generation in which the best-of-run individual occurred (middle column) and the average depth of the population (right column). Each circle corresponds to one run.

4.3. Polynomials

It can be seen in Figure 6 that increasing the degree of the random polynomials results in a behaviour that is characteristic to increased difficulty. If we look at the *rate* of improvement, it is decreasing as the degree increases. The higher degree instances cause a higher rate of code growth and populations with deeper trees. Again, the easier instance (degree 3) has higher edit distance diversity than the harder instance (degree 11). As noted before, the degree 7 polynomial tends to perform with less uniformity than the binomial-3 instances, with a continued high rate of code growth, and lowest edit distance diversity, but with expected fitness and depth. We shall now focus our analysis on one major difference between the results of the binomial-3 and the polynomial experiments regarding the initial decrease in entropy. While the initial fluctuation of entropy is not very important for our final analysis, it demonstrates an interesting difference between the problems.

In Figure 5, we can see that an increase in instance difficulty results in less of an initial decrease in entropy. However, in Figure 6 a smaller decrease in initial entropy can be observed for the lower degree polynomial. As the ERC range is increased in the binomial-3

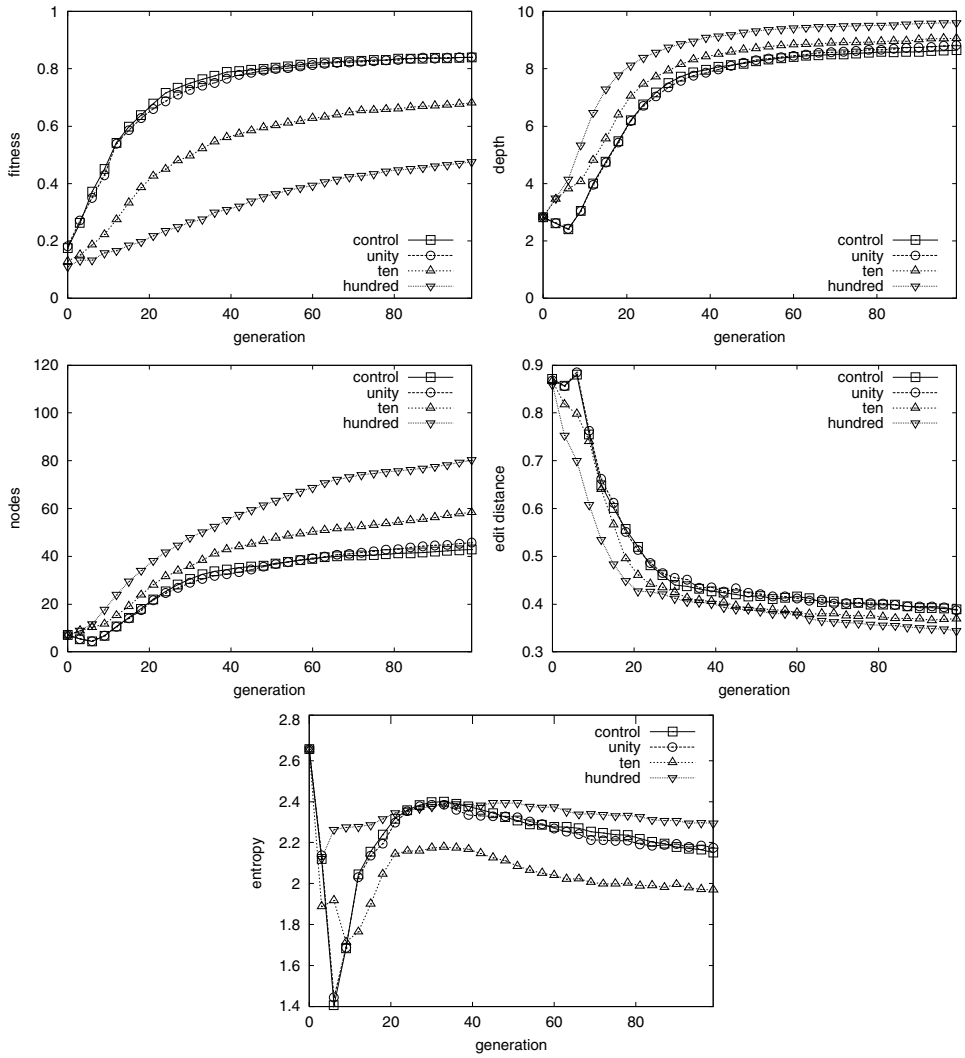

Figure 5. The binomial-3 problem fitness, depth, nodes, edit distance and entropy versus generation for the control, unity, ten, and hundred experiments. Points represent the average over 100 runs.

problem, it is likely that initial populations will be able to represent more unique fitness values than with smaller ERC ranges. While each ERC range does contain the same number of usable constants, some of these values become functionally identical, especially small numbers that may cause division by zero or be similar because of precision representation. The initial greater loss of entropy by the degree 11 polynomial shows the initial diffficulty genetic programming has in representing different initial solutions when the function represents a straight line more closely. It is important to also mention that we are not referring to the initial populations alone, as they all have the same entropy. Rather, it is the populations that undergo the initial rounds of selection that have higher or lower decrease in entropy.

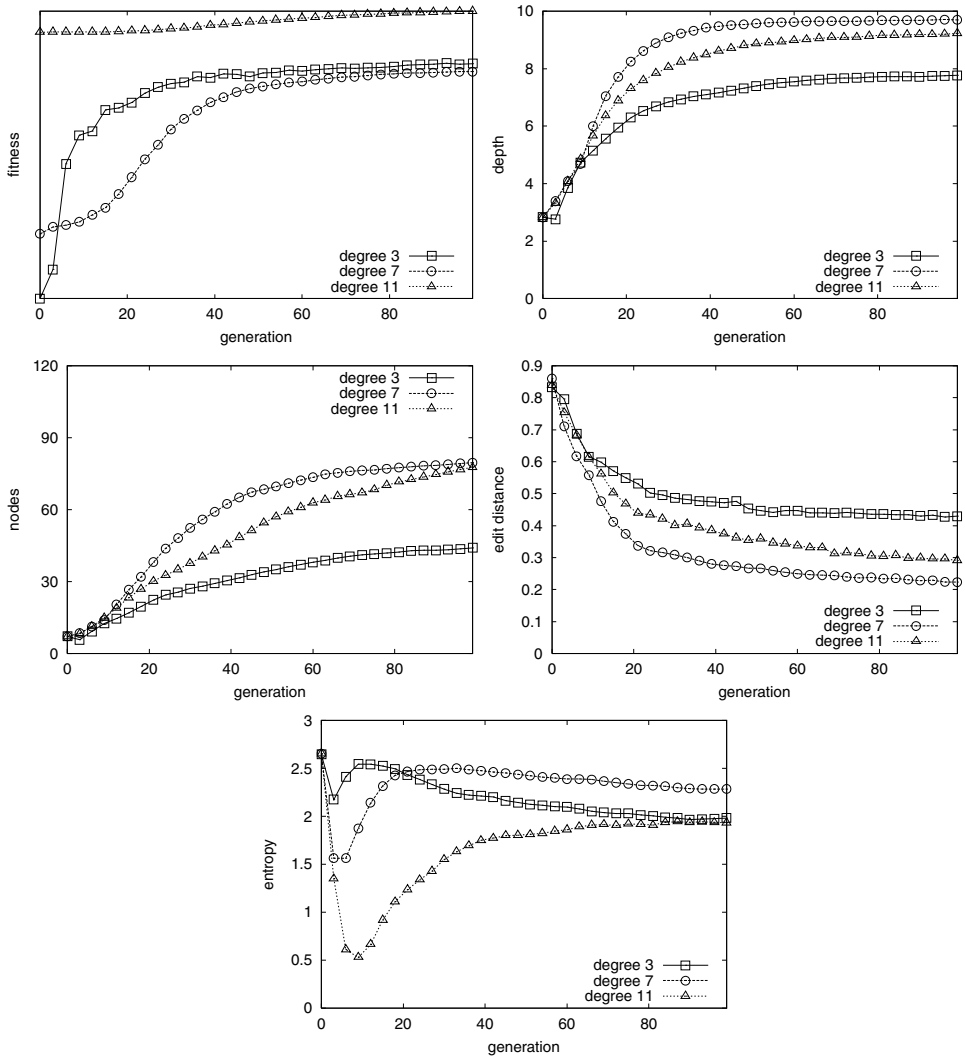

Figure 6. The polynomial problem fitness, depth, nodes, edit distance and entropy versus generation for degree 3, 7 and 11 polynomials. Points represent the average over 100 runs.

5. Discussion

The increase in size, depth and time needed for best-of-run solutions with increased difficulty is clear for both the binomial-3 and polynomial problems. Also, for both problems, many small good solutions are found in early and late generations. Increased size is expected in later generations, but runs appear to have very different rates of growth. What causes the varied rates of code growth?

First, by observing the behaviour of entropy in Figures 5 and 6, we can see that the harder instances contain longer periods of higher (binomial-3) or increasing (polynomials)

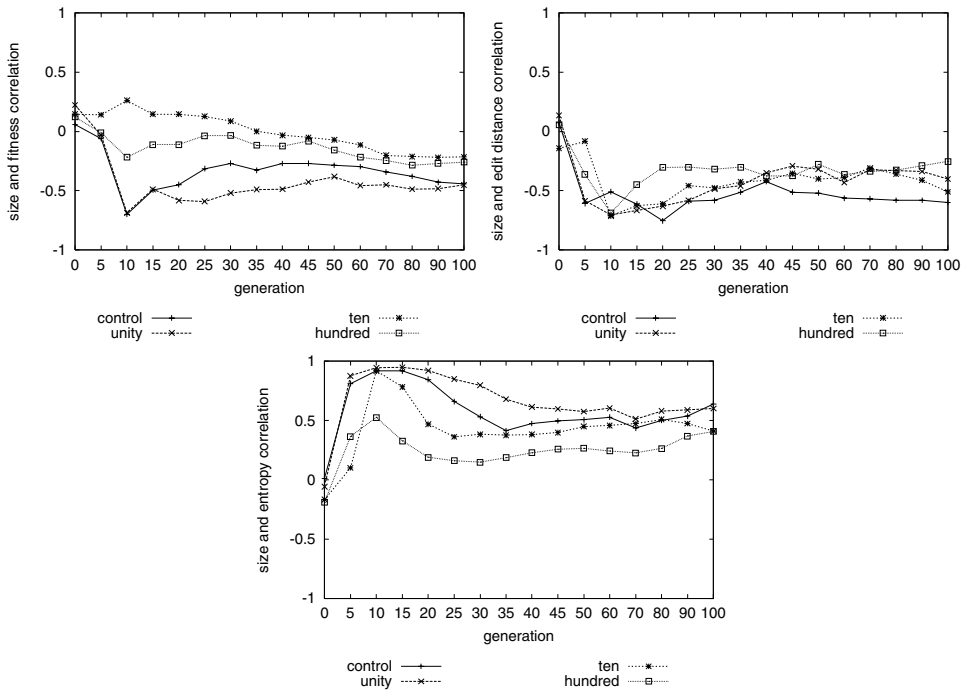

Figure 7. The binomial-3 problem: the Spearman correlation between size and fitness, edit distance and entropy, respectively.

entropy. As entropy describes the number of different fitness values and their distribution in the population, lower entropy will cause selection to become more random. When selection is faced with a population of identical fitness values (the extreme of low entropy), selection becomes purely random. Thus, lower or decreasing entropy in the easier instances will induce a lower selection pressure.

In Figures 7 and 8 we show the evolution of the Spearman correlation coefficient between the average size of a population and its best fitness (raw fitness, where lower is better), entropy and diversity, respectively. We calculate the correlation in every 5th generation up to generation 50, then every 10th afterward. Generally, size is negatively correlated with fitness (low fitness with large size), negatively correlated with edit distance diversity (low edit distance with large size) and positively correlated with entropy (high entropy with large size). There is a clear trend of bigger individuals in populations with higher entropy and lower edit distance.

The 7-degree polynomial is the exception with erratic correlation between edit distance and size. The 7-degree polynomial appears to contain aspects of both the 3-degree polynomial and the 11-degree polynomial. For the 7-degree polynomial there are very few good individuals initially, and they are subsequently overselected (similarly to the degree-11). After the initial period of code-growth, optimal solutions are represented easily by many different programs, leading to reduced entropy and then code growth (as with degree-3), but lower diversity following initial over-selection.

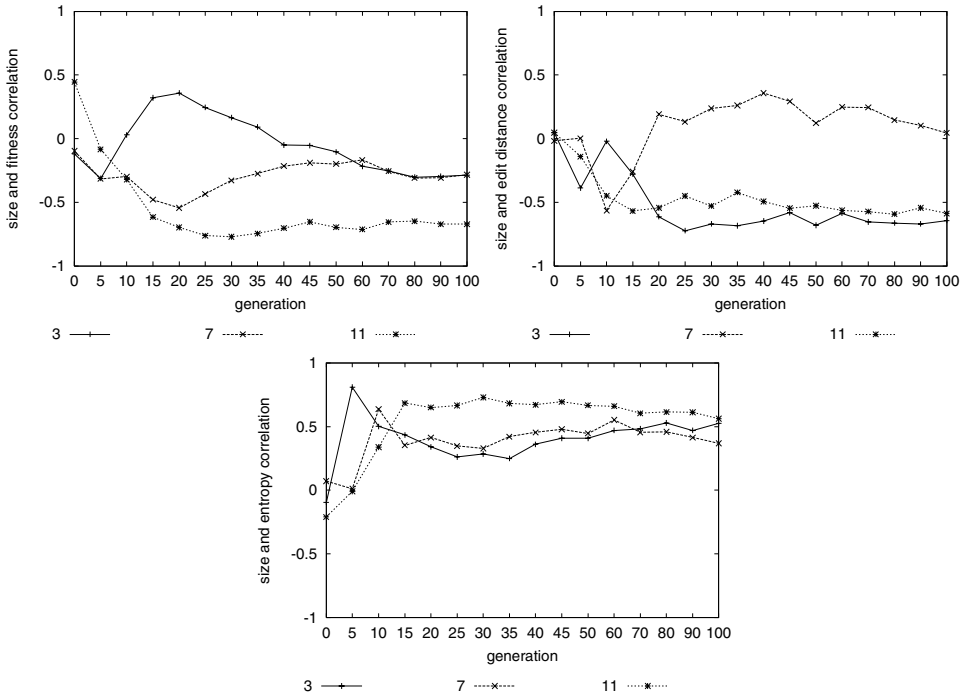

Figure 8. The polynomials: the Spearman correlation between size and fitness, edit distance and entropy, respectively.

5.1. Hypothesis

The increase in instance difficulty makes it more difficult to find good programs, and, when the population converges, to find one or more similarly good (optimal or sub-optimal) programs as well. In the case of easier instances, entropy decreases when the population converges toward individuals with the same optimal (or sub-optimal) fitness. In harder instances, the lack of this behaviour causes higher entropy, which does not decrease as quickly, causing higher selection pressure. This increased selection pressure causes more code growth and also induces less diversity, which causes more code growth, too. This hypothesis of the causal relationship between difficulty, entropy, diversity and growth is shown in Figure 9. The experiments show evidence of higher entropy, less diversity and a higher rate of code growth for harder instances.

We now present supporting evidence for the conjectures that higher selection pressure and lower diversity cause an increased rate of code growth.

Does high selection pressure cause growth? Tackett suggested that the rate of growth is proportional to the amount of selection pressure [34] and Langdon and Poli [15] showed that removing selection pressure stops code growth. To see the effect of varied selection pressure on code growth, we can look at different selection schemes and objective functions

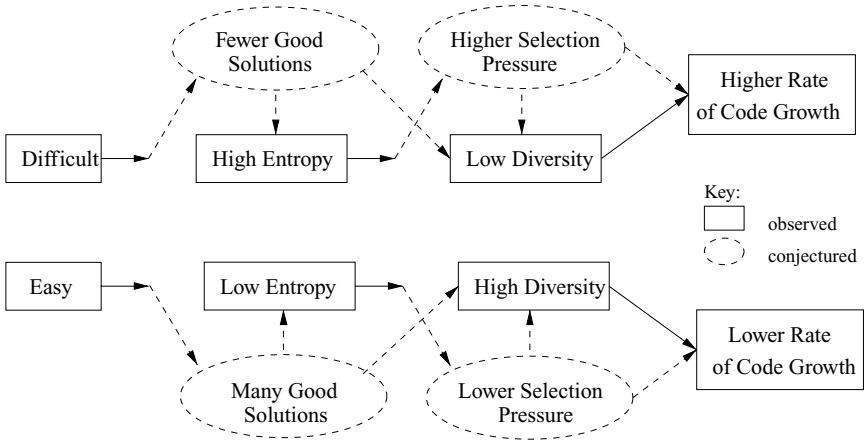

Figure 9. The symmetrical hypothesis that difficulty effects the rate of code growth by maintaining higher levels of entropy, which causes less diversity by the over-selection of better individuals, and both cause more similar individuals, which are likely to be big and to create bigger offspring.

used in the literature. Smith and Harries [31] investigated selection schemes and variable tournament sizes to show the same trend: less selection pressure generally leads to reduced code growth. Poli [28] studied a method for reducing code growth by periodically worsening the fitness of above average sized individuals. In the control experiment with no code growth control, the results (Figure 1, p. 211) show trends of smaller tournament sizes producing less growth.

Considering the theories of code growth, an increased rate of growth with higher selection pressure is expected. Given that the algorithm is biased to produce bigger individuals (as bigger children are more likely to have a higher chance of survival), then the repeated selection of these individuals will consistently produce offspring which in turn have better chance of survival.

Does low diversity cause growth? As the population becomes more similar in shape and content, selection will mate more similar individuals. From the above argument, whether these individuals happen to be the best fit or not, a higher rate of growth will generally occur when the individuals are the same size. Additionally, when the individuals are similar in both size and shape, they will exchange similar size subtrees to produce their offspring. According to the theories of code growth, neither offspring will then be biased to have a higher chance of survival. Instead, offspring will tend to be slightly bigger than their parents, encouraging a higher rate of growth.

In a diverse population (in terms of size, shape and contents), the offspring are more likely to be of varied size, smaller and larger than parents, respectively. Growth will be slower in this case. A higher selection pressure could reverse the effect of diversity by over-selecting the best individuals.

Assuming code growth generally occurs with fitness improvement, can we be confident that the difference in difficulty is suffcient to explain the difference in entropy, diversity and

rate of growth? The regression problems contain possible conflicts between content and context, noise introduced by the protected division operator and a wide range of intron and neutral code effects that could lead to varied rates of growth. Are we sure that our causal hypothesis, described in Figure 9 captures the important aspects that lead to an increased rate of code growth and are not influenced by these factors?

5.2. A model of difficulty

To assess the results in a model without biases introduced by a specific problem domain and fitness function, we create the following model:

1. Trees contain only 'dummy' functions and terminals which are not used to calculate fitness.
2. Fitness is assigned arbitrarily of tree shape or content. All individuals have an initial fitness of 1.0 (0.0 is the best fitness).
3. An offspring which is the same size or larger than its parent is awarded a small fitness improvement, an offspring which is smaller than its parent is penalised by a small amount. This incorporates the theories and corresponding empirical evidence of code growth.

The model consists of binary trees that are initially full with 7 nodes. The algorithm is presented below (we use population size of 100, tournament selection size of 3, and only standard crossover as recombination):

```
The Model Algorithm:
   1. Generate initial population and assign uniform fitness of 1.0
   2. FOR each generation DO
   3.    FOR each individual DO
   4.       assign with probability P the best fitness of 0.0
   5.    FOR the size of the populations DO
   6.       use tournament selection to find two parents
   7.       create a single offspring
   8.       assign the offspring the fitness of the root-parent
   9.       IF the offspring is smaller than the root-parent THEN
   10.         add 0.05 to the fitness
   11.      ELSE
   12.         subtract 0.05 from the fitness, with a minimum of 0.0
```

The *P* value indicates the hardness of the problem. A smaller value denotes the unlikelihood of good solutions being found often and by many individuals (a difficult problem). The higher probability corresponds to the situation where many early individuals can easily represent good solutions and in each generation more new individuals with the same good fitness can be found (an easy problem). The fact that same size or larger individuals are rewarded represents the general consensus of the community in that growth is a condition

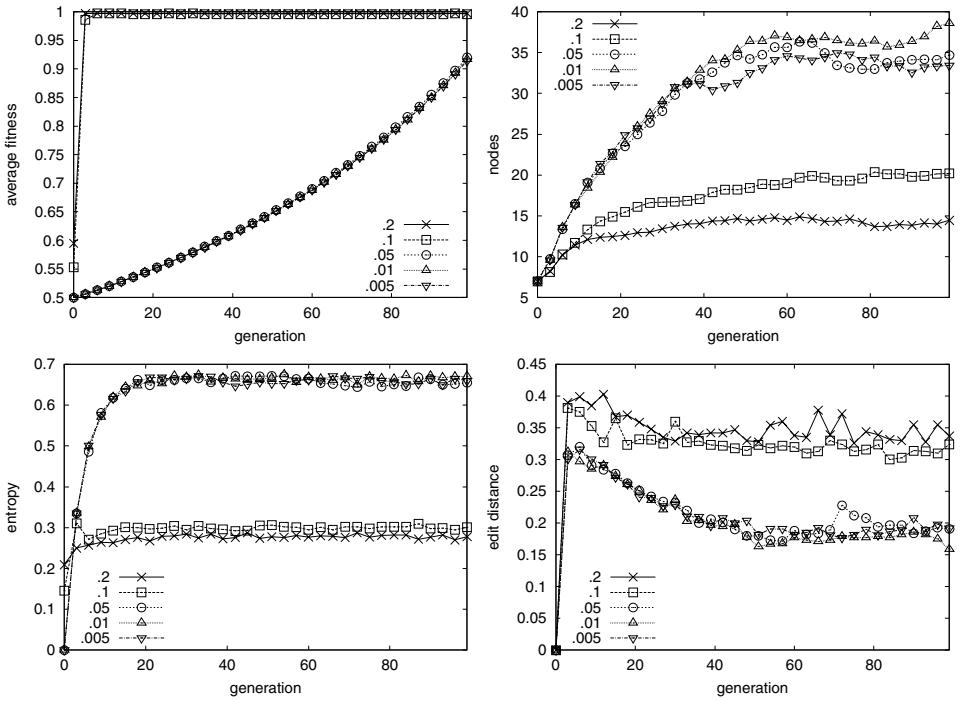

Figure 10. The results on the difficulty model. The values of the probability term P are .2, .1, .05, .01 and .005.

of improved performance in this canonical algorithm. It is the *rate of growth* which we investigate here. Note that the assignment of ideal fitness is independent of size.

We use a range of P values and show a marked difference between easier and harder problems in Figure 10. Easier instances induce a lower entropy and more diversity, which both contribute to a slower rate of code growth. This effect is similar to the results on symbolic regression instances presented earlier. More difficult instances induce higher entropy and lower diversity that dually contribute toward faster code growth. These results support our previous conjecture that reducing the difficulty of instances in this way leads to lower selection pressure (via entropy) and more diversity (due to less selection pressure). These factors cause more different programs to be recombined to produce less code growth.

5.3. Recommendations

The relationship between size and fitness is complex. Previous research has investigated the external dependencies with nodes and the node-to-node interactions in regression problems. Shifting nodes between programs is likely to result in semantic changes during crossover. An increase in program size may help to buffer against detrimental semantic changes. However, where size is not a necessary part of the fitness function, an increase in size

results in a computational cost which becomes an ever-increasing burden on simulation time. Additionally, the many attempts to reduce size that result in weaker fitness illustrate the negative consequences of directly pressuring size.

However, high rates of growth do not seem to be necessary for solving more difficult instances, as many small good solutions are still found. Rather, an increased rate of growth appears to be the effect of higher selection pressure and lower diversity.

Methods which remove large individuals or pressure the population away from increased growth directly are likely to have negative effects. In these cases, good fitness is accumulating in the neighbourhood of large individuals, and by biasing the search away from them, overall performance could be worsened. Therefore, we believe that methods which work to reduce the rate of code growth more gradually will be more effective. By adaptively controlling selection pressure (either directly with the selection measure or via fitness distributions) and improving the diversity of populations, a lower rate of code growth could be consistently achieved.

6. Conclusions and future work

This paper investigates the relationship between difficulty and varied rates of code growth. Two problems with different types of difficulty and a simplified model of genetic programming have been used to understand this relationship. A causal hypothesis is formed that links difficulty to higher entropy, which in turn causes more selection pressure and loss of diversity. Increased selection pressure and lower diversity is then responsible for a higher rate of code growth. This hypothesis is consistent with previous research. A recommendation for controlling diversity and entropy is made to gradually effect the rate of code growth, in response to more direct methods which often have negative effects.

In the future we plan to investigate such methods to effect the rate of code growth without worsening fitness. Testing additional problem domains will help to confirm this hypothesis in the general case. We hope to use these results to build a more clear understanding of the internal workings of genetic programming.

Acknowledgments

The authors would like to thank the anonymous reviewers. S.G. acknowledges the beneficial discussions with Una-May O'Reilly, Bill Hart and Natalio Krasnogor. A.E. acknowledges the support of the School of Computer Science at the University of Birmingham, UK while being a lecturer there.

References

1. D. S. Burke, K. A. De Jong, J. J. Grefenstette, C. Loggia Ramsey, and A. S. Wu, "Putting more genetics into genetic algorithms," Evolutionary Computation, vol. 6, no. 4, pp. 387–410, 1998.
2. E. Burke, S. Gustafson, G. Kendall, and N. Krasnogor, "Advanced population diversity measures in genetic programming", in Seventh International Conference on Parallel Problem Solving from Nature, J. J Merelo, Guervós et al. (Eds.), Springer: Granada, Spain, vol. 2439 of LNCS, 2002, pp. 341–350.

3. J. M. Daida, R. R. Bertram, S. A. Stanhope, J. C. Khoo, S. A. Chaudhary, O. A. Chaudhri, and J. A. Polito II, "What makes a problem GP-hard? analysis of a tunably difficult problem in genetic programming," Genetic Programming and Evolvable Machines, vol. 2, no. 2, pp. 165–191, 2001.
4. E. D. de Jong, R. A. Watson, and J. B. Pollack, "Reducing bloat and promoting diversity using multi-objective methods," in Proceedings of the Genetic and Evolutionary Computation Conference, L. Spector et al. (Eds.), Morgan Kaufmann: San Francisco, CA, 7–11 July 2001, pp. 11–18.
5. A. Ekárt, "Shorter fitness preserving genetic programs," in Artificial Evolution. 4th European Conference, AE'99, Selected Papers, C. Fonlupt et al. (Eds.), Dunkerque, France, 3–5 November 2000, vol. 1829 of LNCS, pp. 73–83.
6. A. Ekárt and S. Z. Németh, "A metric for genetic programs and fitness sharing," in Proceedings of the European Conference on Genetic Programming, R. Poli et al. (Eds.), Springer-Verlag: Edinburgh, 2000, vol. 1802 of LNCS, pp. 259–270.
7. A. Ekárt and S. Z. Németh, "Maintaining the diversity of genetic programs," in Proceedings of the 5th European Genetic Programming Conference, J. Foster et al. (Eds.), Springer-Verlag: Kinsale, Ireland, 3–5 April 2002, vol. 2278 of LNCS, pp. 162–171.
8. A. Ekárt and S. Z. Németh, "Selection based on the Pareto nondomination criterion for controlling code growth in genetic programming," Genetic Programming and Evolvable Machines, vol. 2, no. 1, pp. 61–73, 2001.
9. D. E. Goldberg and U.-M. O'Reilly, "Where does the good stuff go, and why? How contextual semantics influence program structure in simple genetic programming," in Proceedings of the First European Workshop on Genetic Programming, W. Banzhaf et al. (Eds.), Springer-Verlag: Paris, 1998, vol. 1391, pp. 16–36.
10. H. Iba, H. de Garis, and T. Sato, "Genetic programming using a minimum description length principle," in Advances in Genetic Programming, Kenneth E. Kinnear, Jr. (Eds.), MIT Press, 1994, chap. 12, pp. 265–284.
11. C. Igel and K. Chellapilla, "Investigating the influence of depth and degree of genotypic change on fitness in genetic programming," in Proceedings of the Genetic and Evolutionary Computation Conference, W. Banzhaf et al. (Eds.), Morgan Kaufmann: Orlando, Florida, USA, 13-17 July 1999, pp. 1061–1068.
12. J. R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection," MIT Press: Cambridge, MA, USA, 1992.
13. W. B. Langdon, "Quadratic bloat in genetic programming," in Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer (Eds.), Morgan Kaufmann: Las Vegas, Nevada, USA, 10–12 July 2000, pp. 451–458.
14. W. B. Langdon, "Size fair and homologous tree genetic programming crossovers," Genetic Programming and Evolvable Machines, vol. 1, no. 1/2, pp. 95–119, 2000.
15. W. B. Langdon and R. Poli, "Fitness causes bloat: Mutation," in Proceedings of the First European Workshop on Genetic Programming, Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty (Eds.), Springer-Verlag: Paris, April 1998, vol. 1391 of LNCS, pp. 37–48.
16. W. B. Langdon and R. Poli, Foundations of Genetic Programming, Springer-Verlag, 2002.
17. W. B. Langdon, T. Soule, R. Poli, and J. A. Foster, "The evolution of size and shape," in Advances in Genetic Programming 3, Lee Spector et al. (Eds.), MIT Press: Cambridge, MA, USA, June 1999, chapt. 8, pp. 163–190.
18. S. Luke, "ECJ: A java-based evolutionary comp-utation and genetic programming system", 2002. http://www.cs.umd.edu/projects/plus/ecj/.
19. S. Luke, "Modification point depth and genome growth in genetic programming," Evolutionary Computation, vol. 11, no. 1, pp. 67–106, 2003.
20. S. Luke and L. Panait, "Fighting bloat with nonparametric parsimony pressure," in Parallel Problem Solving from Nature PPSN VII, J.-J. Merelo Guervos et al. (Eds.), number 2439 in Lecture Notes in Computer Science, Springer-Verlag: Granada, Spain, 7-11 September 2002, LNCS, p. 411.
21. S. Luke and L. Panait, "Lexicographic parsimony pressure", in Proceedings of the Genetic and Evolutionary Computation Conference, W. B. Langdon et al. (Eds.), Morgan Kaufmann Publishers: New York, 9–13 July 2002, pp. 829–836.
22. M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator".
23. N. F. McPhee and N. J. Hopper, "Analysis of genetic diversity through population history," in Proceedings of the Genetic and Evolutionary Computation Conference, W. Banzhaf et al. (Eds.), Morgan Kaufmann: Florida, USA, 1999, pp. 1112–1120.

24. U.-M. O'Reilly, "Using a distance metric on genetic programs to understand genetic operators," in IEEE International Conference on Systems, Man, and Cybernetics, Computational Cybernetics and Simulation, Florida, USA, 1997, vol. 5, pp. 4092–4097.
25. U.-M. O'Reilly, "The impact of external dependency in genetic programming primitives," in Proceedings of the 1998 IEEE World Congress on Computational Intelligence, IEEE Press: Anchorage, Alaska, USA, 5–9 May 1998, pp. 306–311.
26. U.-M. O'Reilly and D. E. Goldberg, "How fitness structure affects subsolution acquisition in genetic programming," in Genetic Programming 1998: Proceedings of the Third Annual Conference, J. R. Koza et al. (Eds.), Morgan Kaufmann: University of Wisconsin, Madison, Wisconsin, USA, 22–25 July 1998, pp. 269–277.
27. U.-M. O'Reilly and F. Oppacher, "Hybridized crossover-based search techniques for program discovery," in Proceedings of the World Conference on Evolutionary Computation, IEEE Press: Perth, Australia, 29 November–1 December 1995, vol. 2, pp. 573–578.
28. R. Poli, "A simple but theoretically-motivated method to control bloat in genetic programming," in European Conference on Genetic Programming, C. Ryan et al. (Eds.), Springer-Verlag: Essex, 14–16 April 2003, vol. 2610 of LNCS, pp. 200–210.
29. J. Rosca, "Analysis of complexity drift in genetic programming," in Genetic Programming 1997: Proceedings of the Second Annual Conference, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), Morgan Kaufmann: Stanford University, CA, July 1997, pp. 286–294,
30. J. P. Rosca, "Entropy-driven adaptive representation," in Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, J. P. Rosca (Ed.), Tahoe City, California, USA, 9 July 1995, pp. 23–32.
31. P. W. H. Smith and K. Harries, "Code growth, explicitly defined introns, and alternative selection schemes," Evolutionary Computation, vol. 6, no. 4, pp. 339–360, 1998.
32. T. Soule and J. A. Foster, "Effects of code growth and parsimony pressure on populations in genetic programming," Evolutionary Computation, vol. 6, no. 4, pp. 293–309, 1998.
33. T. Soule and R. B. Heckendorn, "An analysis of the causes of code growth in genetic programming," Genetic Programming and Evolvable Machines, vol. 3, no. 3, pp. 283–309, 2002.
34. W. A. Tackett, "Recombination, selection, and the genetic construction of computer programs," PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA, 1994.
35. B.-T. Zhang and H. Mühlenbein, "Balancing accuracy and parsimony in genetic programming," Evolutionary Computation, vol. 3, no. 1, pp. 17–38, 1995.