A Simple but Theoretically-motivated Method to Control Bloat in Genetic Programming

Riccardo Poli
Department of Computer Science, University of Essex

Abstract. This paper presents a simple method to control bloat which is based on the idea of strategically and dynamically creating fitness “holes” in the fitness landscape which repel the population. In particular we create holes by zeroing the fitness of a certain proportion of the offspring that have above average length. Unlike other methods where all individuals are penalised when length constraints are violated, here we randomly penalise only a fixed proportion of all the constraint-violating offspring. The paper describes the theoretical foundation for this method and reports the results of its empirical validation with two relatively hard test problems, which has confirmed the effectiveness of the approach.

1 Introduction

The study of bloat, or code growth, has been one of the most active areas of Genetic Programming (GP) research over the last 10 years [2, 18, 20, 31, 26, 17, 9, 13, 11, 16, 30]. To date several theories propose to at least partially explain bloat. For the sake of brevity here we will briefly review only some of them.

The replication accuracy theory [18, 2, 20] is based on the idea that an important component of the success of a GP individual is its ability to reproduce accurately, i.e., have offspring that are functionally similar to the parent. This would suggest that a GP system will evolve towards representations that increase replication accuracy, all other things being equal. One way this can happen in GP is through the evolution of large blocks of inactive code.

The removal bias theory [29, 17] is based on the observation that the inactive code in a GP tree tends to be in the lower parts of the tree and therefore forms smaller-than-average subtrees. Crossover events which excise one of such inactive subtrees will produce offspring with exactly the same fitness as (one of) their parents. But because, on average, the inserted subtree is bigger than the excised one, the offspring will tend to be bigger than their parents. Towards the end of a run (when producing improved individuals is difficult), when an active node is hit by crossover, the offspring will be worse than their parents more often than not. Therefore, these offspring will tend not to survive. As a result, offspring produced by neutral crossover events will make the biggest contribution to the new generation, but because they are on average bigger than their parents, this will lead to bloat.

The nature of program search spaces theory [17] is based on the experimental observation (later corroborated by strong theoretical evidence [16, Chapter 8]) that, for a variety of test problems (e.g., [15, 9]), above a certain problem dependent size, the distribution of fitnesses does not vary a great deal with the size of the programs. Since
there are more long programs, the number of long programs of a given fitness is greater
than the number of short programs of the same fitness. Thus, over time GP will sample
longer and longer programs simply because there are more of them.

In recent work [19], [23] a quantitative framework has been proposed within which
other, more qualitative theories of bloat could be included. Because this is the starting
point for the method for bloat control we propose in this paper, we will devote a later
section to this theory and its implications. Before we do that, we want to summarise
some of the main techniques for limiting code bloat proposed in the literature.

One of the first and most widely used methods to control bloat is to set a fixed limit
on the size or depth of the programs [8]. Programs exceeding the limit are discarded
and a parent is kept instead. This technique may be effective at limiting code bloat but has
various drawbacks. For example, the limit can interfere with searches once the average
program size approaches the size limit [4, 14]. Also, this approach gives an undesirable
evolutionary advantage to individuals which are likely to produce offspring violating the
depth limit. Effectively programs closer to the size/depth threshold will start replicating
faster than shorter programs with the same fitness. So, once programs start hitting the
threshold, the threshold will start acting as an attractor for the population and will often
cause the population to bloat until it reaches the threshold. A better alternative is to give
a zero fitness to the offspring of above-threshold size. This prevents them from being
used to create the next generation.

In the parsimony pressure method a term is added to the fitness function which penalises
larger programs, thereby encouraging the evolution of smaller solutions. Commonly
the penalty is a simple linear function of the solution size, but other approaches have also been used [6, 31, 1]. Some studies have shown a degradation in performance
when parsimony pressure is used [7, 20]. Recent research suggests that the effect of
parsimony pressure depends on the magnitude of the parsimony function relative to the
size-fitness distribution of the population [26, 28].

Another approach to reducing code bloat has been to modify the basic operators.
One can, for example, vary the rate of crossover (and mutation) to counter the evolution-
ary pressure towards protective code [25], vary the selection probability of crossover
points by using explicitly defined introns [21], or negate destructive crossover events
[27, 22, 5]. Each of these approaches has the goal of reducing the evolutionary
importance of inactive code. Another alternative is to use the class of size-controlling operators defined in [10, 12].

A recent idea to combat bloat is the use of strategies borrowed from the field of
multi-objective optimisation [3, 24]. The idea is to consider fitness as one objective
and size as a second one. Then it is possible to modify selection to use the Pareto
non-domination criterion, thereby giving a reproductive advantage to shorter programs
everything else being equal.

Many of the techniques proposed to combat bloat, as noted in [17], are ad hoc,
preceding rather than following from knowledge of the causes of bloat. In this paper
we will present a very simple technique to control bloat, which is, however, a direct
result of a theoretical analysis of bloat. The paper is organised as follows: in Section 2
we review the theoretical results which form the basis for this technique, in Section 3
we present our method to control bloat, in Section 4 we describe the experiments we
performed to assess the performance of the method, and, finally in Section 5 we draw
some conclusions.
2 Background

As reported in [23] the expected mean size of the programs at generation \( t + 1 \), \( E[\mu(t+1)] \), in a GP system with a symmetric subtree-swapping crossover operator in the absence of mutation can be expressed as

\[
E[\mu(t+1)] = \sum_l N(G_l)p(G_l, t). \tag{1}
\]

where \( l \) is an enumeration of all the possible program shapes, \( G_l \) is the set of programs with the \( l \)-th shape, \( N(G_l) \) is the number of nodes in programs in \( G_l \) and \( p(G_l, t) \) is the probability of selecting a program of shape \( G_l \) from the population at generation \( t \).

This indicates that for symmetric subtree-swapping crossover operators the mean program size evolves as if selection only was acting on the population. This means that if there is a variation in mean size, like for example in the presence of bloat, that can only be attributed to some form of positive or negative selective pressure on some or all the shapes \( G_l \). This can be readily seen by noting that the mean size of the individuals in the population at time \( t \) can be written as

\[
\mu(t) = \sum_l N(G_l)\Phi(G_l, t) \tag{2}
\]

where \( \Phi(G_l, t) \) is the proportion of individuals of shape \( G_l \) in the population at time \( t \). Direct comparison of Equations 1 and 2 tells us that there can be a change in mean program length only if

\[
E[\mu(t+1) - \mu(t)] = \sum_l N(G_l)(p(G_l, t) - \Phi(G_l, t)) \neq 0.
\]

Obviously, this can only happen if the selection probability \( p(G_l, t) \) is different from the proportion \( \Phi(G_l, t) \) for at least some \( l \). So, because \( \sum_l p(G_l, t) = 1 \) and also \( \sum_l \Phi(G_l, t) = 1 \), for bloat to happen there will have to be some short \( G_l \)'s for which \( p(G_l, t) < \Phi(G_l, t) \) and also some longer \( G_l \)'s for which \( p(G_l, t) > \Phi(G_l, t) \) (at least on average). The condition \( p(G_l, t) > \Phi(G_l, t) \), e.g., implies that there must be some members of \( G_l \) which have an above average fitness.

Starting from the results summarised above, in [19] we studied the behaviour of a linear GP system both in the absence and in the presence of fitness. Particularly relevant for this paper was the case where we can assume that the fitness function only has two values, 1 and \( 1 + \bar{f} \). If \( \{B_g\} \) is a set of linear GP programs of uniform fitness \( 1 + \bar{f} \) (which we will call a “hole” or a “spike” depending on the sign of \( \bar{f} \)), and the rest of the search space has fitness 1, then for standard crossover, no mutation, fitness proportionate selection, and for “holes”, i.e., \( \bar{f} < 0 \), we have

\[
\mu(\{B_g\}, t) > \mu(t) \iff E[\mu(t+1)] < \mu(t)
\]
\[
\mu(\{B_g\}, t) < \mu(t) \iff E[\mu(t+1)] > \mu(t),
\]

where \( \mu(\{B_g\}, t) \) is the mean length of the programs in \( \{B_g\} \). (The converse is true for “spikes”.) Thus if the fitness of the \( B_g \) is better than the fitness of the rest of the search
space, the average size of the population will move towards the average size of the $B_p$. If, on the other hand, the fitness of the $B_p$ is worse than the fitness of the rest of the search space, then the average size of the population will move away from the average size of the $B_p$. This observation made us suggest that the creation of artificial fitness holes might be a viable mechanisms to control bloat — a mechanism, however, which was never precisely defined nor tested.

3 The Tarpeian method to control bloat

The results with the “holes” landscapes reported in the previous section suggest that if one could create artificial holes in areas of the search space which we want to avoid, like for example those containing long, bloated programs, then evolution would try to stay away from those areas. This is effectively what happens already in some methods to control bloat. For example, if one zeroes the fitness of individuals which are longer or deeper than a fixed threshold, this corresponds to creating a large fitness hole which discourages the population from getting too close to the threshold. The use of a parsimony pressure reducing the fitness of individuals proportionally to their size also corresponds to creating a hole in the fitness landscape: the only difference being that the hole has smooth, sloping edges.

Here we propose a method of controlling bloat based on the notion of creating fitness holes, but where, unlike previous methods, the holes are created dynamically and non-deterministically. We will call this the Tarpeian method to control bloat, from the Tarpeian Rock in Rome, which in Roman times was the infamous execution place for traitors and criminals. They would be led to its top and then hurled down. The similarity is that, in this method to control bloat, some otherwise normally fit individuals, which are, however, excessively big (and, therefore, traitors), are effectively killed by pushing them down into a fitness hole (the rock’s base).

Before we introduce the method in more detail, we want to rewrite $E[\mu(t+1) - \mu(t)]$ in a form that will allow us to justify the idea of controlling bloat with fitness holes also for the case of fitness functions with more than two values. Let us start by splitting the set of all possible program shapes $\{G_i\}$ into three subsets $G_{<} = \{G_i : N(G_i) < \mu(t)\}$, $G_{=} = \{G_i : N(G_i) = \mu(t)\}$ and $G_{>} = \{G_i : N(G_i) > \mu(t)\}$. Then observe that

$$\mu(t) = \sum_{i} \mu_i p(G_i, t)$$

since $\sum_i p(G_i, t) = 1$. Therefore, we have

$$E[\mu(t+1) - \mu(t)] = \sum_{g \in G_{<}} (N(g) - \mu(t)) p(g, t) + \sum_{g \in G_{=}} (N(g) - \mu(t)) p(g, t) + \sum_{g \in G_{>}} (N(g) - \mu(t)) p(g, t).$$

Thus, the mean program size will be expected to grow if

$$\sum_{g \in G_{>}} (N(g) - \mu(t)) p(g, t) > \sum_{g \in G_{<}} (\mu(t) - N(g)) p(g, t).$$

So, clearly in order to prevent bloat, one needs to either decrease the selection probability $p(g, t)$ for the programs that are bigger than average or increase $p(g, t)$ for the
programs of below-average size, or both. In this paper we will explore the former idea.
(Note, however, that whenever one modifies artificially the selection probability for a part of the population, unavoidably the selection probability for the rest of the population will also be affected. This is because selection probabilities add up to 1.)

In either case, one needs to act on the selection component of the GP system. Reducing the selection probability of a set of programs with specific features does not mean, however, to reduce the selection probability of all the programs in the set nor to do so deterministically, which is exactly what most methods to reduce bloat do (with the notable exception of Ekar’s Pareto-based selection [3]). All is needed is to reduce the selection probability of some of the individuals in the set some of the time. This can be done by directly changing the fitness of the those individuals, or indirectly like in [3].

In the Tarpeian method we randomly select a fixed proportion of the individuals in the population among those whose size is above the current average program size and we set their fitness to the lowest possible value available. This effectively means that those individuals will not be used to create the next generations (and therefore it is as if they had been killed), except in very pathological conditions.

All it is needed to implement the Tarpeian method is a wrapper for the fitness function like the one in the following algorithm, were \( n \) is an integer \( (n \geq 2) \) indicating how frequently the algorithm kills longer-than-average individuals for the purpose of creating a bloat-controlling fitness hole, \( 1/n \) being the proportion of individuals belonging to the hole:

```c
// Tarpeian wrapper
IF size(program) > average_pop_size AND random_int MOD n = 0 THEN return( very_low_fitness ); ELSE return( fitness(program) );
```

An important feature of this algorithm is that it does not require a priori knowledge of the size of the potential solutions to a problem. If programs need to grow in order to improve fitness, the Tarpeian method will not prevent this. It will occasionally kill some individuals that, if evaluated, would result in being fitter than average and this may slow down a little the progress of a run. However, because the wrapper does not evaluate the individuals killed, very little computation is wasted. Even at a high anti-bloat intensity (e.g., when \( n = 2 \)) a better-than-average longer-than-average individual has still a good chance of making it into the population. If enough individuals of this kind are produced (w.r.t. the individuals which are better-than-average but also shorter-than-average), eventually the average size of the programs in the population may grow. However, this growth will still be under control because the Tarpeian method will immediately move the fitness hole so as to discourage further growth. Typically as a run progresses finding individuals with higher-than-average fitness becomes harder and harder. Then, the pressure of the last fitness hole created by the Tarpeian wrapper will make the population shrink. The effect of the parameter \( n \) is not to directly determine the maximum allowed size of the programs in the population. It determines the intensity of the repulsive force exerted by the fitness hole. Once the hole has been jumped, the algorithm recreates one of exactly the same repulsive intensity. That is, the pressure to shrink does not go up as does in the parsimony pressure method. This means that the algorithm will allow growth as long as this is associated to fitness improvements.

A good metaphor to understand the differences between the Tarpeian method and standard parsimony pressure is to imagine that height over the sea level corresponds to
the average population size in a GP run and that the population is like a hot air balloon which is free to fly as high as its content of hot air will allow. Then parsimony pressure behaves like a long elastic band which connects the balloon to earth. The balloon will fly higher and higher, but eventually the force exerted by the elastic band will stop it. The Tarpeian method instead is more similar to gravity, it will not prevent the going up of the balloon. However, gravity is always there waiting for the forces which push up the balloon to weaken. When they do, and they appear to often do this in GP, it makes the balloon go down.

4 Experimental result

To test the Tarpeian algorithm we have considered two problems: the even 10 parity function (which we will call Even-10 in the following) and a symbolic regression problem (which we will call Poly-10) where the target function is the 10-variate cubic polynomial $x_1 x_2 + x_3 x_4 + x_5 x_6 + x_7 x_8 + x_9 x_10$. These problems were chosen because, when attacked with small to medium size populations, they are sufficiently hard to induce a period of slow fitness improvement, which normally induces some form of bloat. The second problem was also chosen because in continuous symbolic regression problems, such as this, code growth may be in part attributed to improvements of the fit between the data and the model.

In the experiments we used a symmetric subtree swapping crossover operator in which crossover points in the two parents were chosen independently and with uniform probability. For both problems the terminal set included only the variables $x_1$ through to $x_{10}$. The function set for Even-10 included the boolean functions AND, OR, NOR, NAND, XOR, EQ and NOT. Fitness was the number of entries of the even parity 10 truth table correctly reproduced. For Poly-10 the function set included $+,-,\cdot$ and a form of protected division which returns the numerator if the magnitude of the denominator is smaller than 0.001. Fitness was minus the sum of the absolute values of the errors made over 50 fitness cases. These were generated by randomly assigning values to the variables $x_i$ in the range [-1,1] and then feeding such values in the cubic polynomial mentioned above. In both cases the population was initialised with the standard GROW method with maximum depth 6. The average program sizes at generation 0 were around 10.3 and 13.1 for Even-10 and Poly-10, respectively. Tournament selection was used (with tournament sizes of 2, 5 and 10). Populations sizes of 100, 500, 1000, 5000 were used. Runs lasted 100 generations and were not stopped even if a 100% solution had been found. In the runs with Tarpeian bloat control, on average we reduced the fitness of 1 in $n$ above-average-size individuals, with $n = 2, 3, 5, 10$ and 20. For the even parity problem the reduced fitness was 0, while for the polynomial regression this was $-10^{20}$. For each parameter setting we performed 30 independent runs.

One may have very different objectives when running a GP system, like for example: (a) obtaining one highly fit solution, (b) obtaining a number of highly fit solutions, (c) obtaining highly fit solutions (like in (a) or (b)) which are also very concise, (d) obtaining highly fit solutions (like in (a) or (b)) as quickly as possible, (e) obtaining highly fit solutions (like in (a) or (b)), which are as concise as possible as quickly as possible. In this paper, our objective is to achieve (a) and (d), although as a side effect of (d) we also indirectly get (c). So, in our analyses we will focus on the highest fitness
individual in each generation and on the average size of the programs in the population (since this is directly related to the amount of time their evaluation will require). Instead of reporting separate plots of best-fitness vs. generation and average size vs. generation, we prefer to use average-size vs. best-fitness plots, since they represent in a more direct way the compromise between these two counteracting objectives. We will also use best-fitness vs. run and average-size vs. run plots for more detailed analyses.

We start from analysing the behaviour of runs without bloat control. As shown by Figure 1 in the absence of a mechanism GP populations tend to bloat dramatically (in this and the following figures pop means “population size” and tsize means “tournament size”). As expected smaller populations are outperformed by the larger ones. More interesting is the detrimental effect of increasing the selective pressure, with the runs with tournaments of size 2 outperforming the others.

![Even-10, No Bloat Control](image1)

![Poly-10, No Bloat Control](image2)

**Fig. 1.** Average behaviour of runs without bloat control. Each data point represents the average over 30 independent runs. Each curve contains 100 points, one per generation.
To provide more information on what is happening in single runs, let us concentrate on the case of tournament size 2. Figure 2 shows the average-size vs. run and best-fitness vs. run diagrams for this case and for the two problems. The data are population averages taken at the last generation of each run. For the sake of visualisation clarity runs have been renumbered so as to show the fitness and size data in ascending order. As we can see from these plots, there are ample differences in performance between runs, but most runs result in bloat. When the population includes only 100 individuals, there are a small number of runs where programs grow little (in the case of Even-10) or shrink (in the case of Poly-10) with respect to their initial sizes. Inspection of the original data reveals that these are runs where fitness is also very poor, suggesting that the system got trapped by a deceptive attractor.

Let us now compare these results with what we achieve with Tarpeian bloat-control on. As shown by Figure 3 in the presence of this mechanism programs tend to grow to a much lesser extent, typically being at least one order of magnitude smaller than in the case without bloat control. Again, as expected smaller populations are outperformed by the larger ones, and, again, there is a detrimental effect of increasing the selective pressure, with the runs performed with tournaments of size 2 outperforming the others. However, here we observe a significant difference in terms of performance between populations with different numbers of individuals. When the population size is relatively small compared with the difficulty of the problem at hand, the Tarpeian method leads to a loss in terms of end-of-run best fitness. This happens for populations of 100, 500 and 1000 individuals in the case of Even-10 and for populations of 100 and 500
individuals for Poly-10. The reason for this is that the repulsive pressure exerted by the fitness hole created by the algorithm may push some populations towards a deceptive attractor. However, when populations are sufficiently large to avoid being captured by the attractor, average performance are often increased when Tarpeian bloat control is present (for example, see the results for populations of size 5000).

Fig. 3. Average behaviour of runs with Tarpeian bloat control with $n = 2$.

Again, to provide more information on what is happening in single runs, we concentrate on the case of tournament size 2. Figure 4 shows the average-size vs. run and best-fitnes vs. run diagrams for this case. As we can see from these plots, with Tarpeian bloat control none of the runs leads to bloat. The detrimental effect of Tarpeian bloat control in the case of incorrectly sized populations is particularly evident for Even-10
with populations of 100 individuals, where most runs end up at the deceptive attractor. The problem gets progressively better as the population size is increased, and it disappears completely for populations of 5000 individuals, where the number of runs hitting a 100% fit solution is increased with respect to the case where no mechanism for bloat control is used. In the case of Poly-10, the situation is similar but only the smallest populations seem to get trapped by the deceptive attractor with a high probability. In the other cases, performance difference w.r.t to when bloat control is absent are minor.

Some of the detrimental effects of Tarpeian bloat control observed in the case of small populations can be cured by reducing the pressure to shrink exerted by the algorithm. This can be easily achieved by setting \( n \) to bigger values. As an example, in Figure 5 we show the behaviour of the algorithm on Even-10 when \( n = 3 \). In exchange for slightly bigger average program sizes, here the algorithm provides good performance also with smaller populations, getting trapped with a high probability by the deceptive attractor only for the smallest population size.

The bottom line is that, although some care is needed in choosing the population size, the selective pressure and the intensity of the repulsive force exerted by the Tarpeian algorithm, when these parameters are correctly set (and this is definitely not difficult to do) the rewards provided by the Tarpeian algorithm are big in terms of best fitness achieved, execution time (runs being typically at least one order of magnitude faster) and parsimony of the evolved solutions.

Fig. 4. Run by run behaviour of a GP system with Tarpeian bloat control with \( n = 2 \).
5 Conclusions and future work

Two years ago [19] we wrote:

“One of the most intriguing possible applications of these ideas is the possibility of using artificially created “holes” to control code growth. The two-level fitness theory suggests that one might be able to slow or stop bloat either by lowering the fitness of a (possibly small) set of large individuals, or by raising the fitness of a set of small individuals. There are important issues, such as sampling errors, that would also need to be studied, but one might eventually be able to develop a method that would allow us to control code growth in a way which limits the changes made to the fitness landscape, thereby limiting the introduced bias.”

In this paper we have developed a method of bloat control, the Tarpeian algorithm, which fulfills this aspiration.

Unlike many other methods to control bloat, this algorithm has a strong theoretical foundation, being in fact entirely motivated by the results of theoretical investigations on the search biases induced by the genetic operators and the phenomenon of bloat. In the empirical tests on two relatively hard test problems reported in the paper, the algorithm proved to be remarkably effective at controlling bloat. Because the individuals killed by the Tarpeian algorithm are not evaluated, the algorithm also has the advantage over other anti-bloat methods of reducing the number of fitness evaluations in addition to reducing the number of nodes to be evaluated.

In the future we intend to explore variants of this algorithm. One idea is to treat above average fitness individuals which are longer than average differently from other longer-than-average individuals, e.g. by creating fitness holes of different depths or just by excluding such individuals from the killing. An alternative approach which would definitely be worth exploring is the possibility of controlling bloat using fitness “spikes” in alternative or in addition to fitness “holes”.

Acknowledgements

The author would like to thank the members of the NEC (Natural and Evolutionary Computation) group at Essex for helpful comments and discussion.
References


