A Comparison of Bloat Control Methods for Genetic Programming

Sean Luke sean@cs.gmu.edu
Liviu Panait lpanait@cs.gmu.edu
Department of Computer Science, George Mason University, 4400 University Drive
MS# 4A5, Fairfax, VA 22030, USA

Abstract

Genetic programming has highlighted the problem of bloat, the uncontrolled growth of the average size of an individual in the population. The most common approach to dealing with bloat in tree-based genetic programming individuals is to limit their maximal allowed depth. An alternative to depth limiting is to punish individuals in some way based on excess size, and our experiments have shown that the combination of depth limiting with such a punitive method is generally more effective than either alone. Which such combinations are most effective at reducing bloat? In this article we augment depth limiting with nine bloat control methods and compare them with one another. These methods are chosen from past literature and from techniques of our own devising. Testing with four genetic programming problems, we identify where each bloat control method performs well on a per-problem basis, and under what settings various methods are effective independent of problem. We report on the results of these tests, and discover an unexpected winner in the cross-platform category.

1 Introduction

The use of arbitrary-length representations in evolutionary computation very often presents a special challenge to the search process. This phenomenon, called *bloat*, is the uncontrolled and unbounded growth of individuals in the population, usually independent of sufficiently justified improvements in fitness. Bloat slows the evolutionary search process, consumes memory, and can hamper effective breeding. The result is a kind of Zeno's paradox, with successive generations slowing by so much that a cap is placed on the effective runtime of the evolutionary system.

Bloat occurs in a variety of evolutionary computation representations, including neural networks, automata, and rule sets. Indeed, the earliest known report of bloating (and of approaches to deal with it) involves evolving Pitt-approach rule systems (Smith, 1980). However, the lion's share of papers on the subject have been in the genetic programming (GP) literature, where bloat has proven to be a major problem.

The dynamics of bloat are still not well understood. Some preliminary theoretical work (Langdon et al., 1999; Poli, 2003) offers hints into macro-level depictions of the phenomenon but not an explanation of the mechanism itself. In Section 2 we discuss current models which attempt to explain bloat, including so-called "intron" models which have until recently been the most popular.

Lacking a full theoretical explanation of bloat, the genetic programming literature has relied on a variety of different approaches to dealing with it, each with its own advantages and disadvantages. By far the most popular such technique is restricting breeding to only produce children less than some maximal tree depth. This approach to dealing with the problem was popularized by Koza (1992), where the depth was set to 17. A distant second is a family of *parsimony pressure* methods, where the size of an individual is a factor in its probability of being selected.

In this paper we examine several approaches to bloat control and report on their success in managing population size while retaining reasonable best-fitness-of-run results. The focus of this paper is on methods which are representation-independent, though our experiments use them in a tree-based genetic programming context. Some of these methods (Linear Parametric Parsimony Pressure, Biased Multi-objective Parsimony Pressure, Tarpeian) are versions of existing methods in the literature. Others (Lexicographic Parsimony Pressure and its variants) have been mentioned very occasionally in the literature but have not been studied in detail before. Still others (Double Tournament, Proportional Tournament, The Waiting Room, death by size) are inventions of our own devising.

Bloat control is usually of secondary concern to the experimenter, behind the quality of the solutions discovered. As such, we believe a method may be considered superior to another if it produces fitter individuals, or if it produces individuals at least as fit but using a more parsimonious population size. To achieve a good balance of fitness and parsimony often requires parameter tweaking in a number of these methods; but thankfully as we will show such tweaking is often problem independent, at least in the four benchmark problems we attempted. Thus in many cases we can present the methods together with the parameter settings with which they appear to do their best.

It is helpful to distinguish between two concerns about bloat: first, that the population as a whole will grow and thus consume resources and cause other difficulties; and second, that the *final individual* returned at the end of the run will be excessively large. The second is a useful concern in its own right given the heuristic belief, prevalent in the machine learning community, that more parsimonious solutions tend to generalize better. However, we are only considering the first concern, as we believe that the resource consumption and difficulty in breeding caused by population-level bloat is a primary difficulty facing experimenters. As such our measure of "bloat" will simply be the mean tree size of all individuals generated during the course of an experimental run.

Such representation-independent bloat control methods can often (but not always) outperform plain tree depth restriction. Surprisingly, however, in previous work (for example (Luke and Panait, 2002b)) we discovered that augmenting a representation-independent method with plain tree depth restriction was nearly universally superior to the representation-independent method alone for some relevant range of parameters. This result has held in our later experiments. Thus in this paper we will always consider representation-independent methods in combination with depth limiting.

We will compare these "combined" methods against plain depth limiting alone. This may seem somewhat odd, but we do this to provide a minimum standard which the combined method must beat, and because depth limiting is the most common and relevant benchmark in the literature. A method "beats" depth limiting only if it produces individuals of equivalent or better fitness but with significantly smaller tree sizes.

After we have compared the methods with various parameters against depth limiting alone, we then pit them, with their best settings, against one another. We do so by comparing the methods using their best "cross-problem-domain" parameter settings, and also (of less value) using their best "problem-domain-dependent" parameter settings.

2 Why Does Bloat Occur?

The literature is getting closer to a general theory of code bloat, but specific predictions are hard to come by as the dynamics of genetic programming selection, breeding, and evaluation are complex. The high-level general explanation is usually this: the dynamics of the representation and the breeding process are such that adding material to a tree is more strongly correlated (or less negatively correlated!) with fitness improvement than removing material from the tree is. Poli (2003) puts forth an argument which casts this traditional explanation in a theoretical framework, but at present there is no theoretical explanation for *how* or *why* this correlation arises.

Early theories of bloat concentrated on the existence of *introns*, regions of code which performed no function. Specifically, a particular form of intron known as *inviable code* is crucial in that not only does the code perform no function, but that it cannot be replaced with any code which *does* perform a function. Subtree modification in this region permits very large subtrees which do not change the fitness at all. This is due to the presence of some sort of *invalidator*, a structure in the tree which nullifies the entire region regardless. For example, in the Lisp s-expression `(+ 4 (∗ 0 (+ x 9)))` the subexpression `(+ x 9)` is inviable code: the invalidator `(∗ 0 ... )` guarantees that no matter what the inviable code is, its return value will be reduced to 0 regardless.

There are three major intron theories historically. The original theory, *hitchhiking* (Tackett, 1994), simply posits that introns accumulated through attachment to popular building-blocks, and because there was no real need to be rid of them. This theory only suggests a propagation method and not an explanation for why it was more likely that introns would become attached in the first place than to be removed eventually. This led to the more popular second theory, *defense against crossover*, which posited that inviable code was selected for because it made it more difficult to damage the fitness of an individual through a crossover event (more inviable code results in a higher likelihood that crossover would occur in an inviable code region). This would benefit the individual late in the evolutionary process, when fitness gains are hard to be had. This theory has been cited by a wide range of literature, for example, (Blickle, 1996; Nordin and Banzhaf, 1995; Banzhaf et al., 1998; Langdon et al., 1999)— for a more complete listing, see (Luke, 2000, 2003). The third theory, *removal bias*, argues that to take advantage of crossover in inviable code regions, the removed subtree must be limited to the inviable code subtree size at most; but that there is no such requirement for insertion of subtrees, thus creating a fitness bias for insertion (Soule and Foster, 1998b).

These intron theories noted a relationship between the size of trees and the amount of introns in those trees, and suggested a causal relationship (that the growth of introns was causing tree bloat). However Luke (2000, 2003) has demonstrated that the casual relationship is likely the other way around. Invalidator structures are dispersed throughout a tree, but invalidate entire subtrees, and so large trees are likely to have invalidators proportionally closer to the root, thus nullifying a much larger percentage of the tree. Hence larger trees naturally have higher percentages of invalid code, all other things being equal. Furthermore, it is relatively easy to produce an experiment which falsifies the defense against crossover and removal bias theories: simply disallow crossover in inviable code regions (a procedure called *marking*). As it turns out, doing so has relatively little impact on code growth.

One non-intron theory, *fitness causes bloat* (Langdon and Poli, 1997b), suggests that trees grow because there are more highly-fit large trees than small ones, and as the evolutionary process started with small trees, it may simply be moving to equilibrium. We have noted difficulties with this theory, but most importantly its lack of a functional explanation (Luke, 2003). However, the theory accurately predicts a sub-quadratic increase in tree growth.

A final non-intron theory is of our own devising, and we feel that it is best supported by the evidence at present (Luke, 2000, 2003). Through analysis of tree modification statistics with and without marking, it can be shown that a very strong indicator of child survivability (defined as how often the child is selected for breeding) is the depth of the tree-modification point. The intuition is that nodes near the root are in some sense usually more important to the fitness of an individual than ones near the periphery, because nodes near the root filter the final returned results and make the highest-level control decisions. Modification point depth is related to bloat in two ways: first, larger mothers[1] have more deep points, and are also more likely to produce larger children; and second, deeper points are likely to root smaller subtrees, resulting in a removal bias. This theory is applicable with and without inviable code but in its present form only explains bloat in tree-based genetic programming representations.

Gustafson et al. (2004) argue for a relationship between problem difficulty and the rate of code growth if not a specific explanation for how it occurs. The authors link problem difficulty with entropy in fitness distributions, which is in turn linked to selection pressure, which is in turn linked to the rate of code growth.

2.1 Controlling Bloat

Unfortunately this preliminary theoretical work has not yet yielded formally justified bloat-control methods. Lacking this, bloat control techniques are still relatively ad-hoc. One exception is the Tarpeian method (Poli, 2003), which claims theoretical motivation though not actual justification; but we believe this motivation is general and holds for most methods presented here.

The most obvious approach to bloat control, employed in the methods discussed here, is to punish individuals in some way for being large. Many such methods are called *parsimony pressure* methods because this punishment takes the form of selective pressure to be small. But this is not the only kind of punishment: for example, The Waiting Room punishes individuals by delaying their introduction into the population; and the Death by Size method punishes individuals by making them more likely candidates for death in steady-state evolution.

However, the most common bloat-control method used in genetic programming is instead to place constraints on the size or depth of the genetic programming trees. As noted by Luke (2000, 2003) there are many different ways to place such constraints. The usual approach, as done by Koza (1992), rejects children whose depth exceeds 17, placing copies of their parents in the population in their stead. This is the approach we use in this paper. Some other depth limiting variants in the literature: Martin and Poli (2002) retry crossover some N times until it generates a valid-depth child; and Silva and Almeida (2003) dynamically expand the tree depth as fitter individuals are discovered. Depth limiting has been criticized in the past for its effect on breeding (Haynes, 1998), but it has proven a surprisingly successful method (Luke and Panait, 2002b,a).

Some other methods in the literature, not compared here, include:

Explicitly Defined Introns This GP-specific technique allows the inclusion of special nodes which adapt the likelihood of subtree crossover or mutation at specific positions in the tree (Nordin et al., 1996; Smith and Harries, 1998; Blickle, 1996; Angeline, 1996).

[1] A child is produced by removing a subtree from a “mother” and inserting a subtree donated by a “father”.

Code Editing One easy way to attack growth is to directly simplify and optimize an individual's tree. Soule et al. (1996) report strong results with this approach. However, Haynes (1998) warns that editing can lead to premature convergence.

Pseudo-Hillclimbing This technique rejects children if they are not superior to (or simply different from) their parents in fitness. If a child is rejected from joining the next generation, a copy of its parent joins the next generation in its stead. One effect of this technique is to replicate large numbers of parents into future generations; earlier individuals are generally smaller than later individuals (due to bloat), this results in slower growth in average size. This technique has been reported with some success in (Langdon and Poli, 1998; Soule and Foster, 1998b).

Size- or Depth-Determined Genetic Operators Here the genetic operators are chosen at least in part on the size or depth of the tree. Larger trees receive more destructive operators (Kennedy and Giraud-Carrier, 1999).

Population Limits Various hard and soft limits on total population size may be implemented to prevent the population as a whole from growing too large (Wagner and Michalewicz, 2001; Silva and Costa, 2005).

2.2 Effects of Bloat Control

There are various reasons to apply bloat control. First, one might hope that its application will result in a more parsimonious final solution. Second, one might hope that its application will warp the search trajectories to stay in "small tree" regions where good solutions are to be had (depending on problem). Third, one might hope bloat control will increase the total number of evaluations the system can perform in a given time, and will prevent bloat from consuming all available memory. For some problem domains, evaluation cost is not directly related to individual size: but memory is a concern in a great many situations.

In light of the concerns above, genetic programming may be viewed as a race against time, where the objective is to get the best results possible before bloat sets in and makes further progress ineffective. Bloat could plausibly cause this in two ways. First, as the size of trees grow, the speed of the evolutionary system slows so much that at some point further progress is not worthwhile due to the computational cost. But even if one ignores wall clock time, bloat could act as a kind of Mueller's Ratchet, pushing the search trajectory into "large tree space" from which it is difficult to return. However, applying bloat control with a heavy hand presents its own difficulties. If the bloat control is too strong, the search trajectory is biased against large trees even if they are likely to lead to fitter regions of the search space, resulting in a poor final outcome. In the worst case, small trees are preferred regardless of how poor they are in fitness. Bloat control methods thus present a trade-off for the experimenter.

As can be seen in most of the figures later in this paper, as the "strength" of a particular bloat control method increases, the average tree size generally decreases. However, best fitness of run often stays steady until some point at which it starts worsening. If there is such a point, this often provides us with a "sweet spot" for the method: the smallest tree size just before the best fitness of run begins to worsen. Where this point is (or if it exists), and the rate of change in tree size and best fitness of run, varies from method to method and from problem to problem, but the trend is quite strong. While in a few cases certain methods clearly fail for a given domain, in most cases the various methods presented here simply differ in the degree to which they have this sweet spot, and the degree to which this sweet spot is sensitive to the problem domain or other particulars.

One surprising effect we cannot readily explain is the interaction between depth limiting and the various other bloat control methods presented here. In many cases these methods outperformed depth limiting: but surprisingly the *combination* of the two was nearly universally superior to either separately. Why this is the case is still somewhat of a mystery to us, but its universality (and depth limiting's surprising effectiveness) has prompted us to compare the combination against depth limiting alone, rather than compare the methods unaugmented by depth limiting.

3 Experimental Setup

All experiments used population sizes of 1000. The runs did not stop when an ideal individual was found. The plain depth limiting runs used tournament selection of size 7. We chose four problem domains: Artificial Ant, 11-Bit Boolean Multiplexer, Symbolic Regression, and 5-Bit Even Parity. Symbolic Regression used the function $x^4 + x^3 + x^2 + x$, with no ephemeral random constants. Artificial Ant used the Santa Fe food trail, and individuals were permitted to run for 400 timesteps. Further description of these problem domains can be found in (Koza, 1992; Luke, 2000).

It is helpful to note some features of these four problem domains which might effect bloat dynamics. Artificial Ant evolves trees which direct an ant to eat as many food pellets as possible within 400 time steps. Fitness is the number of pellets remaining, and the trail has less than 100 of them, so there are relatively few fitness values an individual may take on. As a result, it is common for distinct individuals to have identical fitness. The 11-Bit Boolean Multiplexer and 5-Bit Even Parity problems both require the individual to learn a complex boolean function, and as such tend to generate fairly large correct solutions. 11-Bit Boolean Multiplexer has integer fitness values ranging from 0 to 2048. It is known that 11-Bit Boolean Multiplexer has relatively little inviable code, but most individuals' fitnesses fall into multiples of 32. 5-Bit Even Parity has the fewest number of fitness values: only integer fitness values ranging from 0 to 32. Symbolic Regression asks trees to fit a real-valued function within the domain [-1,1] but with any valid range; thus individuals can take on any real-valued fitness. However, Symbolic Regression suffers from a very large amount of inviable code, so many individuals in the population have identical fitness.

We set all other parameters to closely follow those in (Koza, 1992). Specifically: we used tournament selection of size 7. New generations were created through repeatedly selecting one individual (with 10% probability) to produce one child through direct copying, or two individuals (with 90% probability) for crossover to produce two children. Subtree crossover points were chosen either (with 90% probability) uniformly from the nonterminal nodes or (with 10% probability) uniformly from the terminal nodes. Initial trees were created using the "ramped half-and-half" algorithm in (Koza, 1992) with a maximal tree depth of six nodes. Identical individuals were disallowed in the initial population. Depth limiting was part of the subtree crossover procedure: if a child was generated with a depth larger than 17, it was replaced with a copy of its parent instead.

Statistical significance was determined by pairwise t-tests, and we adjusted the confidence level for the tests as indicated by Bonferoni's inequality. We performed all t-tests at 95% confidence when comparing fitnesses, and at 99.995% confidence when comparing tree sizes. The reason for the difference was that we wanted to make sure we caught any deterioration in fitness, while the mean tree size should be much smaller to be considered significantly better. In the Symbolic Regression domain, comparisons of fitnesses were performed with ranked tests in order to cope with non-normality. The evolutionary computation system used was ECJ (Luke, 2005). For all four problem domains, lower fitness is better.

4 The Tarpeian Method

The Tarpeian method works by making uncompetitive some fraction W of individuals with above-average size (Poli, 2003). The method was implemented as follows: before the evaluation process, individuals with above-average size (number of nodes) were assigned a very bad fitness (10^{20} in our experiments) with probability W. Such individuals were not evaluated further, and such a low fitness dramatically reduced any chance of an individual being selected for breeding (it was still possible to be selected if the tournament selection procedure happened to select only individuals of this type, but the chances of this happening were slim).

As individuals are marked uncompetitive *before* evaluation, it is not necessary to evaluate them. This is a crucial feature of the Tarpeian method, as it reduces the number of evaluations necessary. This feature makes Tarpeian different from the other methods, which all evaluate every individual generated. Thus to compare this method fairly with others, we ran Tarpeian experiments for 50,000 actual evaluations (equivalent to 50 generations of 1000 individuals each for other methods). This extension of evaluations for Tarpeian is crucial to the success of the method: if one must for some reason evaluate every individual — for use in coevolution techniques, say — then Tarpeian performs poorly in our experiments. We also performed runs using Tarpeian without depth limiting (as done in (Poli, 2003)), and combining with depth limiting was much better.

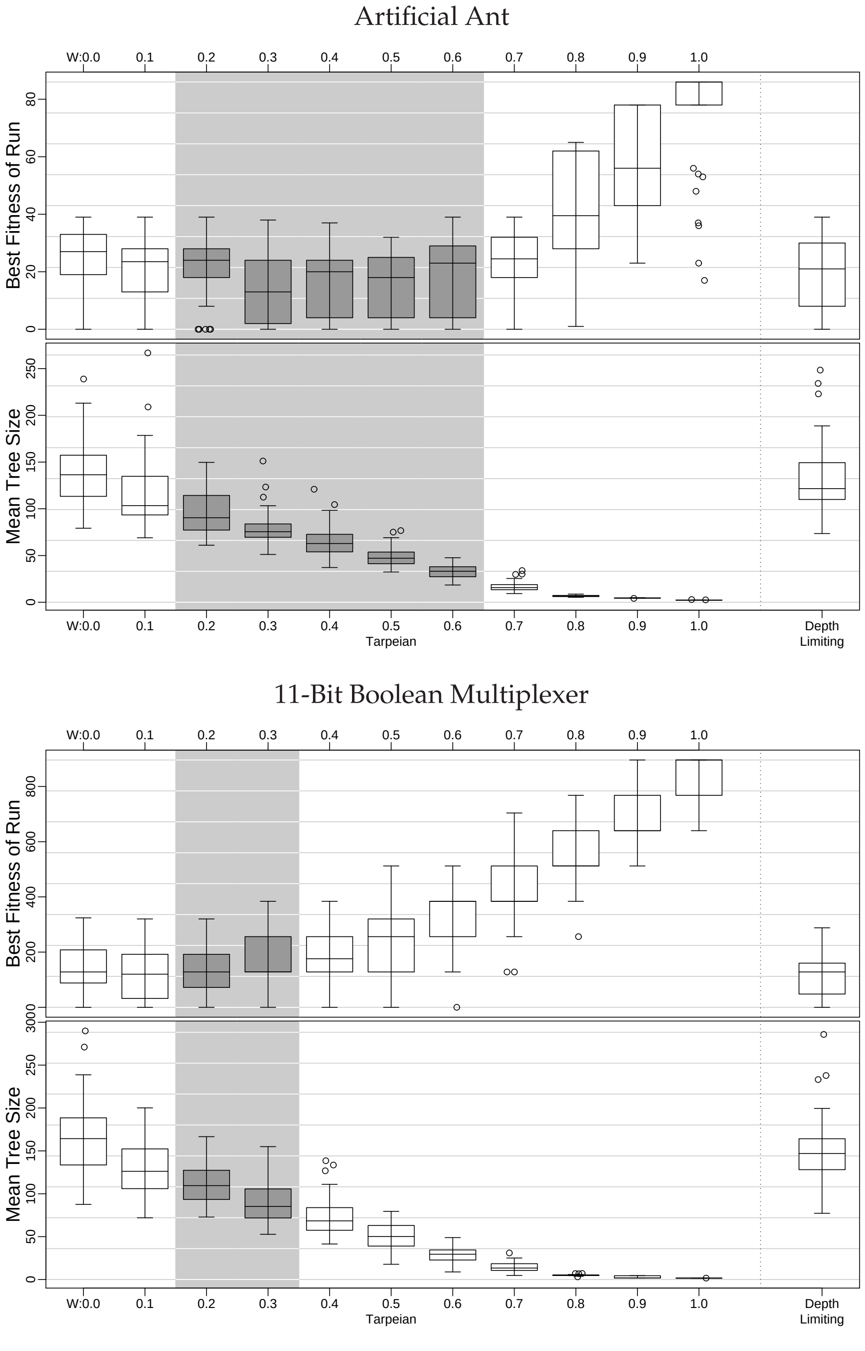
To provide a budget of 50,000 evaluations, we adjusted the number of generations for different values of W to completely consume this budget. Consequently, runs with W=0.0 used 50 generations, while for example an average of 57 generations was necessary for runs with W=0.3 to exhaust their budget. Due to our use of generation boundaries, the total number of evaluations did not exactly match the expected 50,000: in this case, we allowed Tarpeian to run for more evaluations to meet the next generation boundary.

The results of experiments with the Tarpeian method are summarized in Figures 1 and 2. Lower settings for W do not put enough pressure on the tree size to decrease it significantly, but high values of W put far too much pressure on parsimony at the expense of worse fitness. Highlighted in gray are those intermediate settings of W that resulted in fitness as good as depth limiting alone but had a smaller average tree size per run.

The very feature that makes Tarpeian attractive (no need to evaluate individuals unnecessarily) is also its downside: Tarpeian is overly aggressive. The technique rejects individuals by size before considering their fitness, so if W is high, Tarpeian will tend to reject an individual no matter how fit it is. As a result, Tarpeian is unusually sensitive to parameters, though we found one ($W = 0.3$) which was consistently good across all four domains.

5 Linear Parametric Parsimony Pressure

Parsimony pressure is a general family of methods which consider size as part of the selection process. Historically such methods work by computing the fitness g of an individual as a function of the individual's raw fitness f and its size s). We refer to


Artificial Ant

11-Bit Boolean Multiplexer

Figure 1: Boxplot of best fitness and mean tree size of run for the Tarpeian method (in combination with depth-limiting), as compared to Koza-style depth limiting alone. The Tarpeian method stochastically kills a ratio W of above-average individuals; we varied W from 0.0 (no parsimony pressure) to 1.0 (extreme parsimony pressure) in increments of 0.1. The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

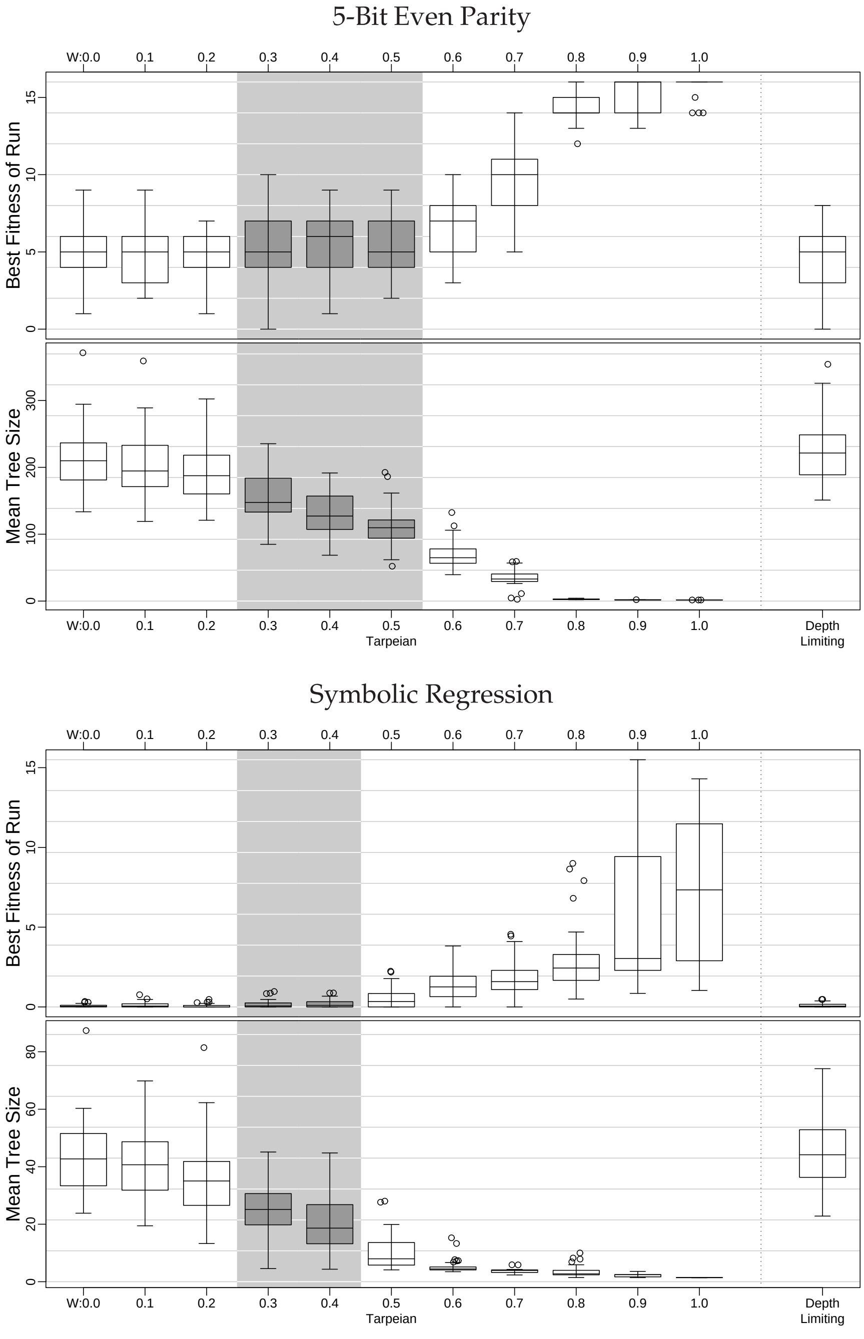


Figure 2: Boxplot of best fitness and mean tree size of run for the Tarpeian method (in combination with depth-limiting), as compared to Koza-style depth limiting alone. The Tarpeian method stochastically kills a ratio W of above-average individuals; we varied W from 0.0 (no parsimony pressure) to 1.0 (extreme parsimony pressure) in increments of 0.1. The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

such methods as *parametric parsimony pressure*, as they result in a fitness model which uses the exact values of f and s as parameters in a statistical model of selection.

The trouble with parametric parsimony pressure is that it essentially defines f as being worth so many units of s. One must tune the parsimony pressure so as not to overwhelm the raw fitness metric. This can be difficult when the fitness assessment procedure is nonlinear, as is usually the case: it may well be that a difference between 0.9 and 0.91 in raw fitness is much more dramatic than a difference between 0.7 and 0.9. Parametric parsimony pressure can thus give size an unwanted advantage over raw fitness when the difference in raw fitness is only 0.01 as opposed to 0.2. Unexpected size-parameter dominance can also arise when the population's raw fitnesses are converging late in the evolution procedure. Notice that these issues are similar to those which gave rise to the preference of tournament selection and other nonparametric selection procedures over fitness-proportionate selection.

The most widely-used approach to parametric parsimony pressure is to treat the individual's size as a linear factor in fitness, that is, $g = xf + ys$, where x and y are parameter settings.[2] This technique has been used in both GP (Koza, 1992) and in non-GP (Burke et al., 1998), and it is the method we consider in this paper. Linear Parametric Parsimony Pressure is occasionally augmented with a limit z, that is if $s \leq z$ then $g = xf$, else $g = xf + y(s - z)$ (Cavaretta and Chellapilla, 1999). Belpaeme (1999) used a similar limit, but considered maximal tree depth rather than size as the parameter. Nordin and Banzhaf (1995) also applied parametric parsimony pressure, believed to be linear, to evolve machine language GP strings. Linear Parametric Parsimony Pressure has been used in combination with adaptive strategies. Zhang and Mühlenbein (1995) adjusted y based on current population quality. Iba et al. (1994) were similar, but used information-theoretic functions for f and s. Linear Parametric Parsimony Pressure has also been applied in stages: first by setting $g = f$, then factoring in size only after the population has reached a sufficient quality (Kalganova and Miller, 1999). Soule and Foster (1998a) present an analysis of Linear Parametric Parsimony Pressure and when and why it can fail.

Some non-GP papers (Wu et al., 1999; Bassett and De Jong, 2000) use a non-linear parametric parsimony pressure: $g = (1 - ys)f$. Bassett and De Jong note that this has the added benefit of increasing the penalty proportionally to the fitness.

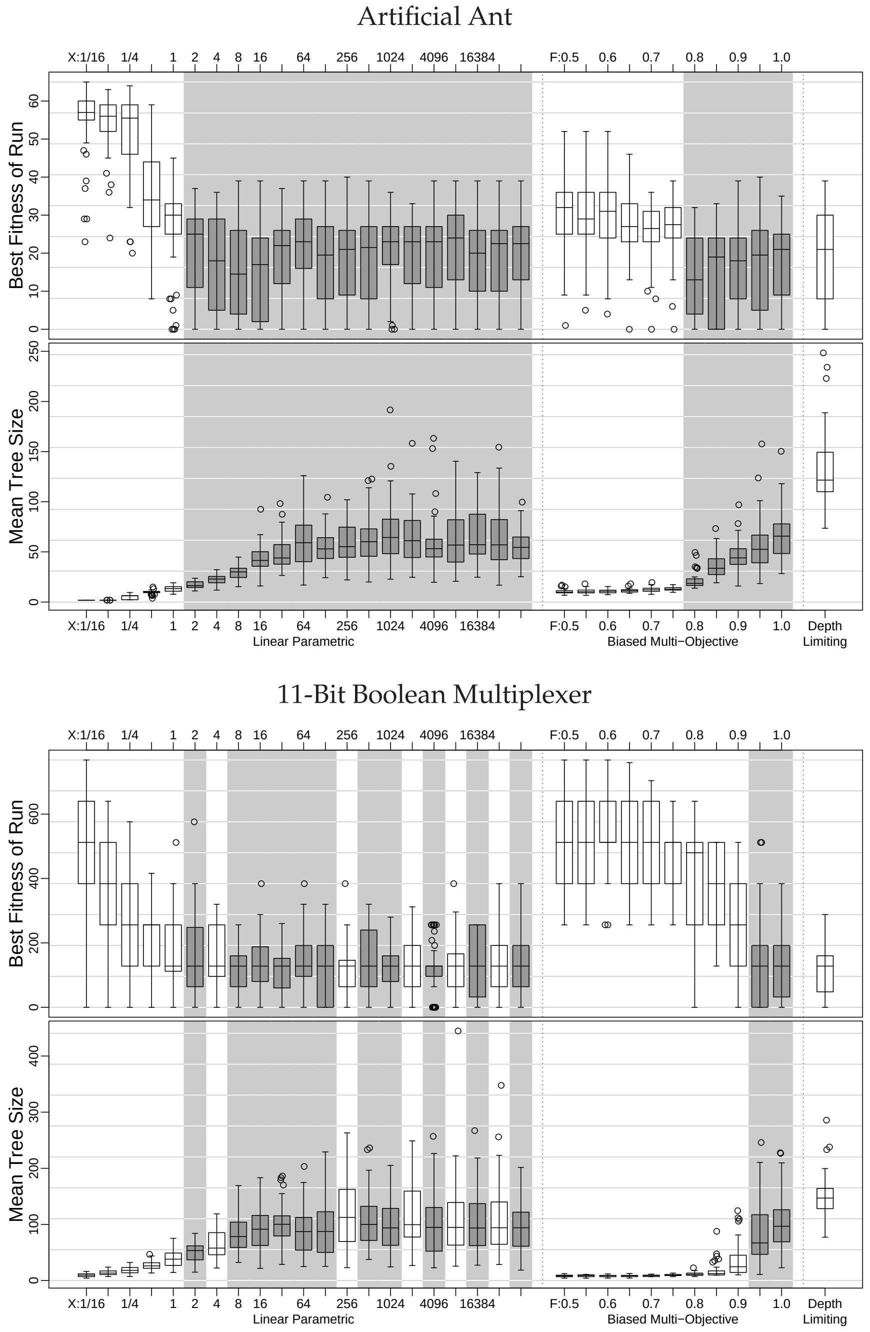
5.1 Experiment

We performed experiments with Linear Parametric Parsimony Pressure of the form $g = xf + s$ (that is, $y = 1$). We expected good settings for x in Linear Parametric Parsimony Pressure to be dependent on the particulars of the fitness function. For this reason we tested all four problem domains over a wide enough range for x to provide decent results for all of them. x was varied from $\frac{1}{16}$ to 65536, doubling x each time.

The results are shown in Figures 3 and 4. As can be seen, the tree size and fitness curves are dependent on the peculiarities of the problem domain; but for these four problems they were not *that* different. Indeed, there existed settings ($x = 32, 64, 128, 512$ and 1024) for which all four problem domains yielded statistically significantly smaller trees while maintaining at least as good fitness as depth limiting alone.

The results in the Artificial Ant and 5-Bit Even Parity problems are very similar: Linear Parametric in combination with depth limiting outperforms depth limiting alone if the weight on fitness is above a minimum threshold. Similar results are

[2] Here we add the size rather than subtract it because in the experiments shown later, lower fitness is better. Later parametric parsimony equations are similarly adjusted.



Artificial Ant

11-Bit Boolean Multiplexer

Figure 3: Boxplot of best fitness and mean tree size of run for Linear Parametric and Biased Multi-objective (in combination with depth-limiting), as compared to Koza-style depth limiting alone. For Linear Parametric, the weight of the fitness component is denoted *X*; the proportion of comparisons based on fitness for Biased Multi-objective is denoted *F*. The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

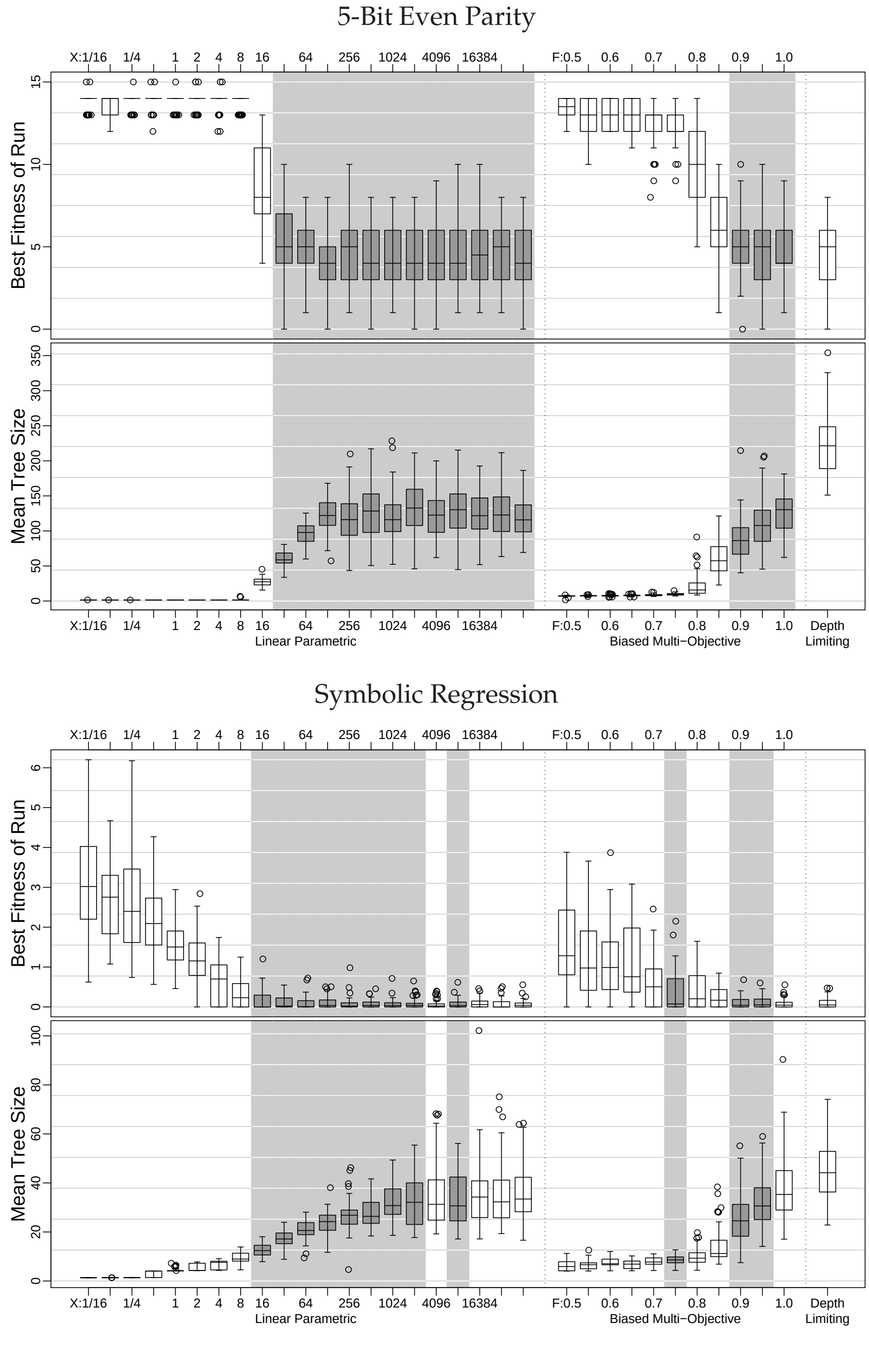


Figure 4: Boxplot of best fitness and mean tree size of run for Linear Parametric and Biased Multi-objective (in combination with depth-limiting), as compared to Koza-style depth limiting alone. For Linear Parametric, the weight of the fitness component is denoted *X*, the weight of the size component is 1. The proportion of comparisons based on fitness for Biased Multi-objective is denoted *F*. The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

obtained in the 11-Bit Boolean Multiplexer domain, except that the significance of the results is very close to our desired confidence level. Symbolic Regression led to a range (16 to 2048) where Linear Parametric Parsimony Pressure was superior.

Unlike Tarpeian, which had minimum and maximum range values, Linear Parametric had a minimum X value for which it was effective, but in most cases one could increase X to arbitrarily large values (that is, smaller and smaller amounts of pressure) and still have reasonable results. The only exception is Symbolic Regression, where Linear Parametric peters out at about $X = 8192$. This counterintuitive result may be explained by the fact that as $X \to \infty$, Linear Parametric approaches Lexicographic Parsimony Pressure which, for reasons discussed later, is effective in all the tested problem domains except for Symbolic Regression.

6 Pareto-based Multi-objective Parsimony Pressure

An alternative to parametric methods is to consider size and fitness as separate objectives in a multi-objective selection procedure. Some literature uses a Pareto-dominance optimization scheme to do this (Bleuler et al., 2001; de Jong and Pollack, 2003; Ekart and Nemeth, 2001). In such a scheme, individual A is said to *dominate* individual B if A is as good as B in all objectives (fitness, size) and is better than B in at least one objective. This family of methods uses one of several multi-objective optimization algorithms to discover the "front" of solutions which are dominated by no one else.

The literature has had mixed results, primarily because nondominated individuals tend to cluster in the corners of the front (size-only, fitness-only), and a highly fit but poorly sized individual is undesirable, as is a tiny but highly unfit individual. de Jong and Pollack (2003) tackled this problem successfully by introducing diversity preferences into the Pareto system, thereby discouraging these clusters. Similarly Bleuler et al. (2001); Ekart and Nemeth (2001) deal with the issue through the choice of an evolutionary algorithm (SPEA2) custom-built to enforce diversity along the Pareto front.

Here we propose a similar method which can be used by more ordinary evolutionary computation procedures. Rather than promote diversity along the entire Pareto front, our method, called *Biased Multi-objective Parsimony Pressure* (BMOPP), tries to bias the search along the front towards the fitness end of the front.

BMOPP, like other Pareto methods, uses fitness and size as its two objectives. At each generation, BMOPP places each individual into a *Pareto layer* as follows. First, individuals along the nondominated front of the two objectives are assigned to layer 1. Those individuals are then removed from consideration, and a new front is computed from among the remaining individuals. Individuals in that new front are assigned to layer 2 and removed from consideration, another new front is then computed, and so on.

After Pareto layers have been established, individuals are selected using a form of tournament selection which, with probability F, compares individuals based solely on their fitnesses, and with probability $1 - F$ compares them based on their respective Pareto layers (lower layers being preferred). Ties are broken using the alternative comparison (fitness or Pareto layer). If both individuals are identical in fitness and in layer, one individual is chosen at random. The particular value of F is of interest to us: as F increases, selection is more likely to be based on tree size and less likely to be based on fitness. We note that if $F = 1$, BMOPP is exactly plain Lexicographic Parsimony Pressure (the method discussed next). If $F = 0$, BMOPP is entirely Pareto-based.

We note that in some sense most of the algorithms presented here (particularly Linear Parametric, Lexicographic, and Double and Proportional Tournaments) are "multi-objective" in that they are seeking to define fitnesses both in terms of raw fitness and of size. We reserve the term multi-objective here to mean instead those Pareto-based methods which now are prevalent in the multiobjective stochastic optimization field.

6.1 Experiment

We allowed F to range from 0.5 to 1.0, increasing by 0.05 each time. The results are shown together with the Linear Parametric Parsimony Pressure results in Figures 3 and 4. As expected, the range of values where BMOPP was successful tended to be in the upper end of this range; this is in accordance to the finding in (de Jong and Pollack, 2003). The threshold for F where BMOPP significantly improves results depends on the problem domain: 0.8 for Artificial Ant, 0.9 for 5-Bit Even Parity and Symbolic Regression, and 0.95 for 11-Bit Boolean Multiplexer ($F = 0.95$ improved results in all four problem domains).

We observe that for small values of F, populations are allowed to cluster many small individuals near the extreme parsimony end of the Pareto front, pulling resources from search for better fit individuals. This occurs very rapidly: near 0.7 tree size drops off precipitously. And just as with Linear Parametric, we also note that BMOPP has no maximum good F value except in Symbolic Regression, and surprisingly for the same reason! As mentioned earlier, when $F = 1$, BMOPP approaches Lexicographic Parsimony Pressure which, as discussed next, fails on Symbolic Regression.

7 Lexicographic Parsimony Pressure

Lexicographic Parsimony Pressure is a straightforward technique for optimizing both fitness and tree size, by treating fitness as the primary objective and tree size as a secondary objective in a lexicographic ordering. The procedure does not assign a new fitness value, but instead uses a modified tournament selection operator to consider size.

Comparison of individuals, as is done during selection, works as follows: an individual is considered superior to another if it is better in fitness; if they have the same fitness, then an individual is considered superior if it is smaller; if they are also the same size, the superior individual is determined at random.

Plain Lexicographic Parsimony Pressure is attractive in that it has nothing to tune. However, the procedure only works well in environments which have a large number of individuals with identical fitness. As it so happens, genetic programming is just such an environment, thanks to a large amount of inviable code and other events causing neutral crossovers and mutations. Of course, there exist problem domains where few individuals have the same fitness. For these domains we propose two possible modifications of Lexicographic Parsimony Pressure, both based on the notion of sorting the population, putting it into ranked buckets, and treating each individual in the bucket as if it had the same fitness. These two modifications are:

Direct Bucketing The number of buckets, b, is specified beforehand, and each is assigned a rank from 1 to b. The population, of size p, is sorted by fitness. The bottom $\lceil p/b \rceil$ individuals are placed in the worst ranked bucket, plus any individuals remaining in the population with the same fitness as the best individual in the bucket. Then the second worst $\lceil p/b \rceil$ individuals are placed in the second worst ranked bucket, plus any individuals in the population equal in fitness to the best individual in that bucket. This continues until there are no individuals in the population. Note that the topmost bucket with individuals can hold fewer than $\lceil p/b \rceil$ individuals, if p is not a multiple of

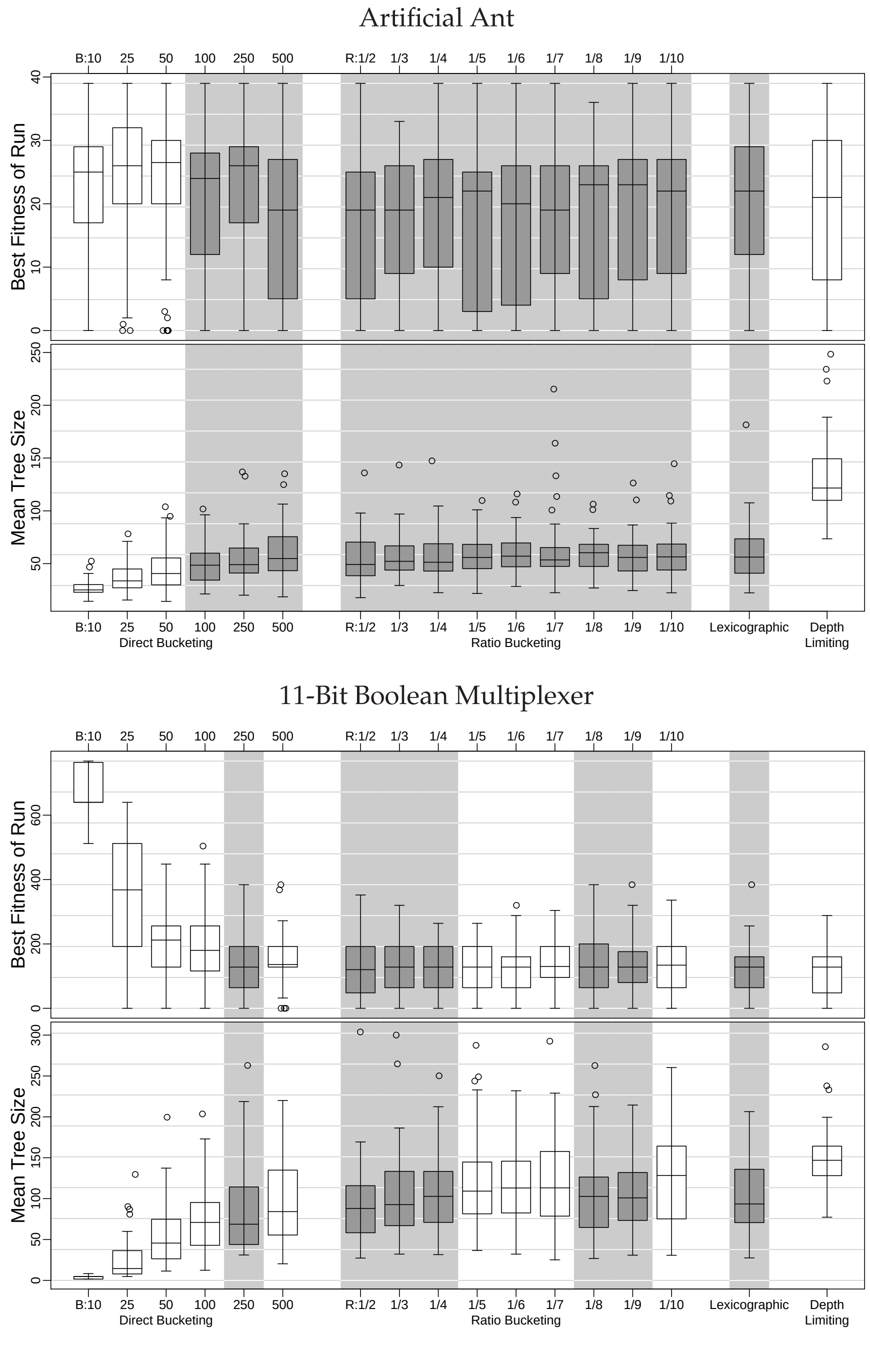


Figure 5: Boxplot of best fitness and mean tree size of run for Direct Bucketing, Ratio Bucketing, and Lexicographic Parsimony Pressure (in combination with depth-limiting), as compared to Koza-style depth limiting alone. The bucket size for Direct Bucketing is denoted *B*, and the ratio for Ratio Bucketing is denoted *R*. The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

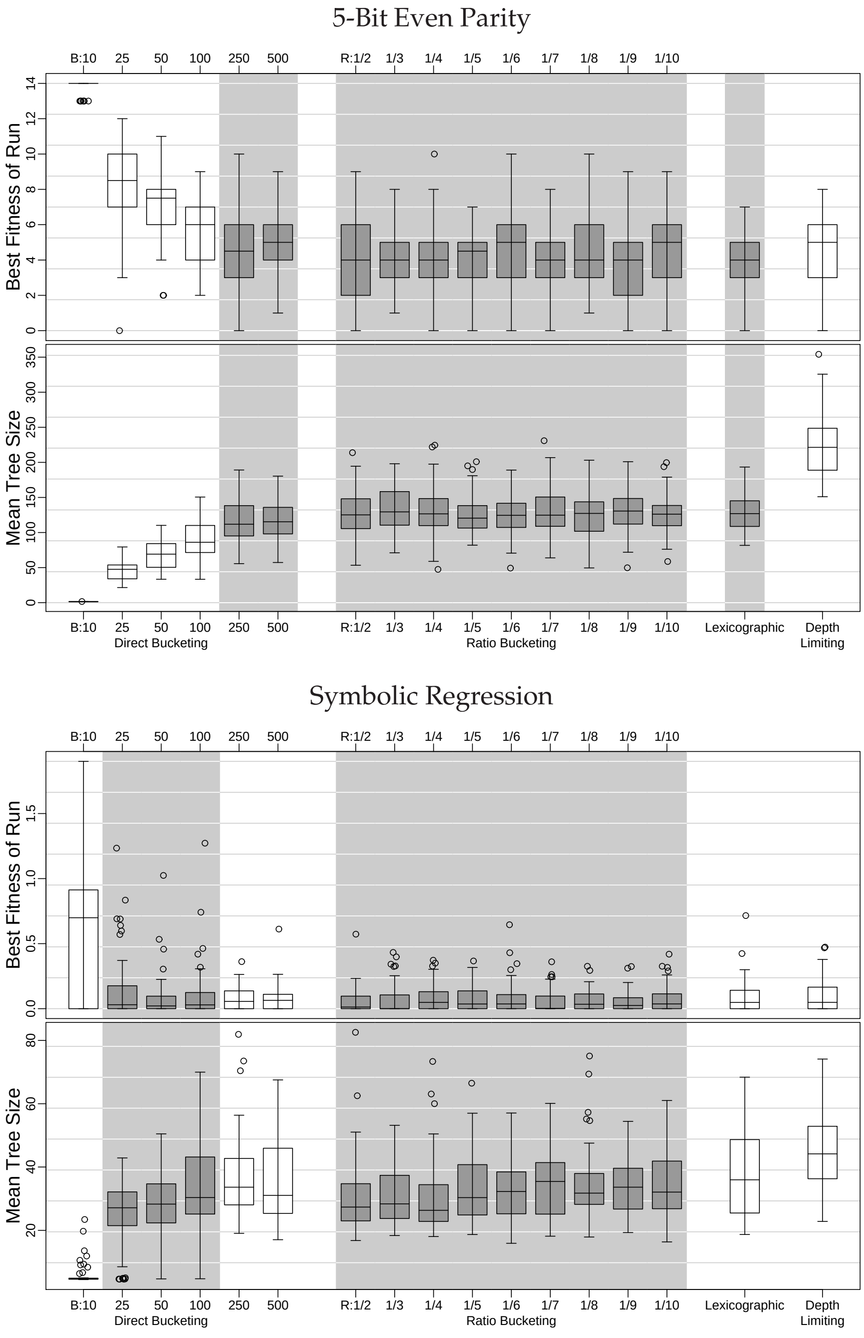


Figure 6: Boxplot of best fitness and mean tree size of run for Direct Bucketing, Ratio Bucketing, and Lexicographic Parsimony Pressure (in combination with depth-limiting), as compared to Koza-style depth limiting alone. The bucket size for Direct Bucketing is denoted *B*, and the ratio for Ratio Bucketing is denoted *R*. The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

b. Depending on the number of equal-fitness individuals in the population, there can be some top buckets that are never filled. The fitness of each individual in a bucket is set to the rank of the bucket holding it. Direct bucketing has the effect of trading off fitness differences for size. Thus the larger the bucket, the stronger the emphasis on size as a secondary objective.

Ratio Bucketing Here the buckets are proportioned so that low-fitness individuals are placed into much larger buckets than high-fitness individuals. A bucket ratio $1/r$ is specified beforehand. The bottom $\lceil 1/r \rceil$ fraction of individuals of the population are placed into the bottom bucket. If any individuals remain in the population with the same fitness as the best individual in the bottom bucket, they too are placed in that bucket. Of the remaining population, the bottom $\lceil 1/r \rceil$ fraction of individuals are placed into the next bucket, plus any individuals remaining in the population with the same fitness as the best individual now in that bucket, and so on. This continues until every member of the population has been placed in a bucket. Once again, the fitness of every individual in a bucket is set to the rank of the bucket relative to other buckets. As the remaining population decreases, the $\lceil 1/r \rceil$ fraction also decreases: hence higher-ranked buckets generally hold fewer individuals than lower-ranked buckets. Ratio bucketing thus allows parsimony to have more of an effect on average when two similar low-fitness individuals are considered than when two high-fitness individuals are considered.

Both bucketing schemes fill the buckets with remaining individuals equal in fitness to the best individual in the bucket. The purpose of this is to guarantee that all individuals of the same fitness fall into the same bucket and thus have the same rank. This removes artifacts due to the particular ordering of the population. Bucketing schemes require that the user specify a bucket parameter (either the number of buckets or the bucket ratio). This parameter guides how strong an effect parsimony can have on the selection procedure. Note however that this parameter is not a direct factor in fitness. Thus the specific difference in fitness between two individuals is still immaterial; all that matters is fitness rank.

Although no doubt Lexicographic Parsimony Pressure has been used in the past, there is surprisingly little literature on the subject, and no real consideration of the method except in passing. Lucas (1994) used a Linear Parametric function to evolve bitstrings used in context-free grammars: but the size was multiplied by a constant small enough to guarantee that the largest possible advantage for small size was less than the smallest difference in fitness. We believe the fitness was then fed into a fitness-proportional selection operator. In the *pygmies and civil servants* algorithm (Ryan, 1994), crossover is always between one “civil servant” and one “pygmy”. A pygmy is selected using Linear Parametric Parsimony Pressure with a heavy weight for small size. A civil servant is selected using plain Lexicographic Parsimony Pressure. The technique was also used in (Olsson, 1995; Nedja et al., 2006). Some recent literature (Silva and Almeida, 2003; Silva and Costa, 2004) compared other methods to Lexicographic Parsimony Pressure in response to our earlier work on the subject (Luke and Panait, 2002b).

We note an interesting relationship between Lexicographic Parsimony Pressure and Linear Parametric Parsimony Pressure. In the Linear Parametric Parsimony Pressure equation $g = xf + ys$, as $\frac{x}{y}$ increases, the equation in some sense approaches plain Lexicographic Parsimony Pressure. This is because when fitness has very large weight relative to size, size matters only when fitnesses are identical.

7.1 Experiment

We compared plain depth limiting against plain Lexicographic Parsimony Pressure, Lexicographic Parsimony Pressure with direct bucketing, and Lexicographic Parsimony Pressure with ratio bucketing, all in combination with depth limiting. Direct bucketing used buckets of size $B = 10$, 25, 50, 100, 250, or 500. Ratio bucketing used bucket ratios of size $R = 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9$, or $1/10$.

In previous work (Luke and Panait, 2002b), we compared these methods *without* the help of additional depth limiting, and discovered that they could produce superior results to depth limiting alone for all problems except for Symbolic Regression. This problem domain is unusual in that very large chunks of code can be added to the bottom of trees and produce small, but non-zero, changes in fitness. Thus Symbolic Regression allows for incremental fitness improvements which present a significant challenge to a lexicographic approach. Because Linear Parametric Parsimony Pressure and BMOPP both approach lexicographic parsimony at the top ends of their parameter values, they too fail on Symbolic Regression at their top ends.

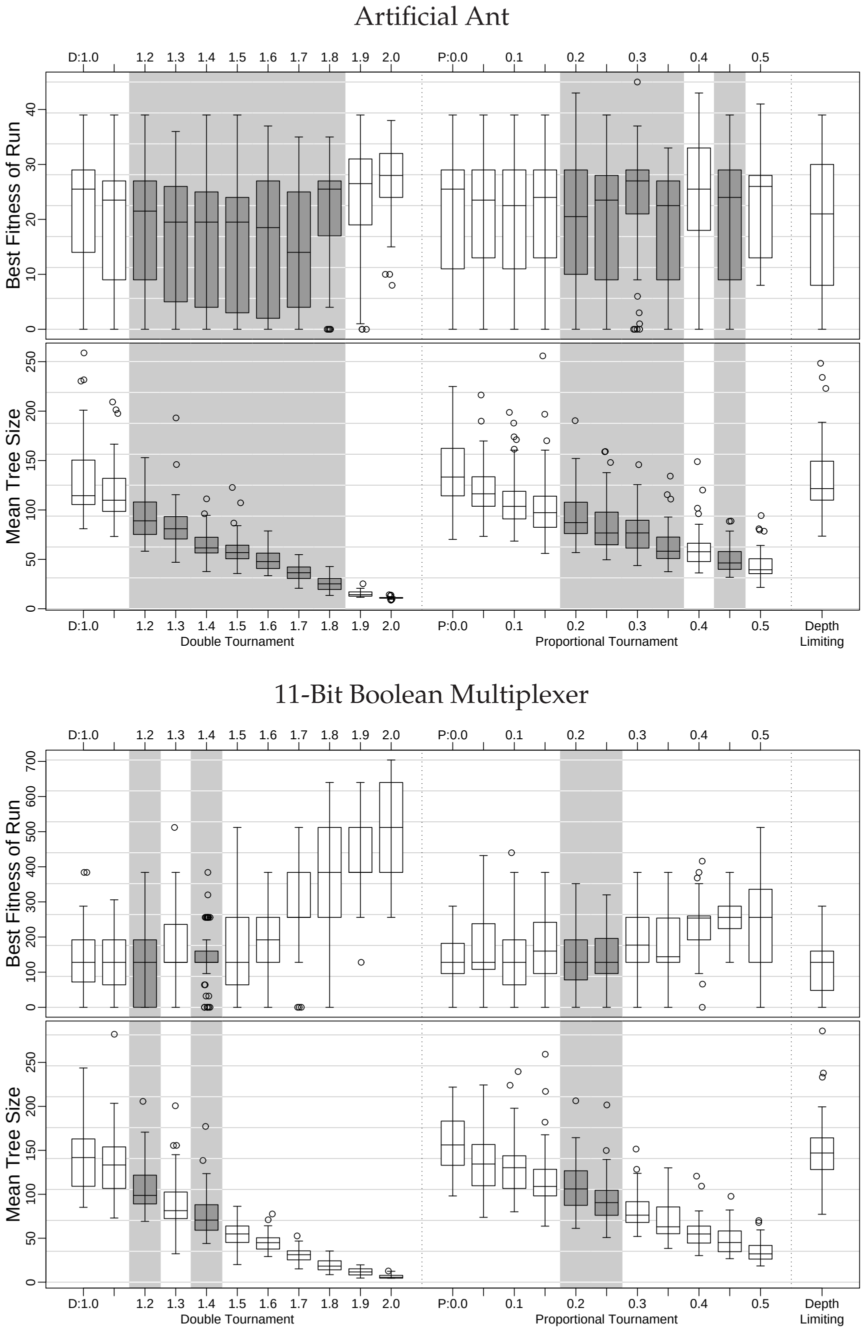
The results are shown in Figures 5 and 6. Even when augmented with depth limiting, plain Lexicographic Parsimony Pressure *still* could not statistically significantly outperform depth limiting alone in the Symbolic Regression problem. Direct bucketing likewise did not have a parameter setting which was consistently good across all problem domains: Symbolic Regression required a different bucket size than the other methods. However, ratio bucketing was nearly uniformly superior in any setting. Consistent values for ratio bucketing included $R = 1/2, 1/3, 1/4, 1/8$, and $1/9$. We do not have an explanation for the peculiar sizes in the Multiplexer problem domain, where the confidence is reduced to only 99% that the results are improved by using ratio bucketing (in $R = 1/5$, $1/6$ and $1/7$) due to high variance.

8 Double and Proportional Tournament

Double and Proportional Tournament are non-parametric methods proposed by us which use variations on tournament selection. Double Tournament applies *two* layers of tournaments in series, first for fitness and then for size (or in the other order). Proportional Tournament flips a coin to determine which objective (fitness or size) to use for a given tournament selection.

Double Tournament The Double Tournament algorithm selects an individual using tournament selection: however the tournament contestants are not chosen at random with replacement from the population. Instead, they were each the winners of *another* tournament selection. For example, imagine if the “final” tournament has a pool size of 7: then seven “qualifier” tournaments are held as normal in tournament selection, and the winners go on to compete in the final tournament. This can be used for parsimony pressure by having the final tournament select based on parsimony while the qualifying tournaments select based on fitness (or vice versa). The algorithm has three parameters: a fitness tournament size F, a parsimony tournament size D, and a switch (*do-fitness-first*) which indicates whether the qualifiers select on fitness and the final selects on size, or (if false) the other way around.

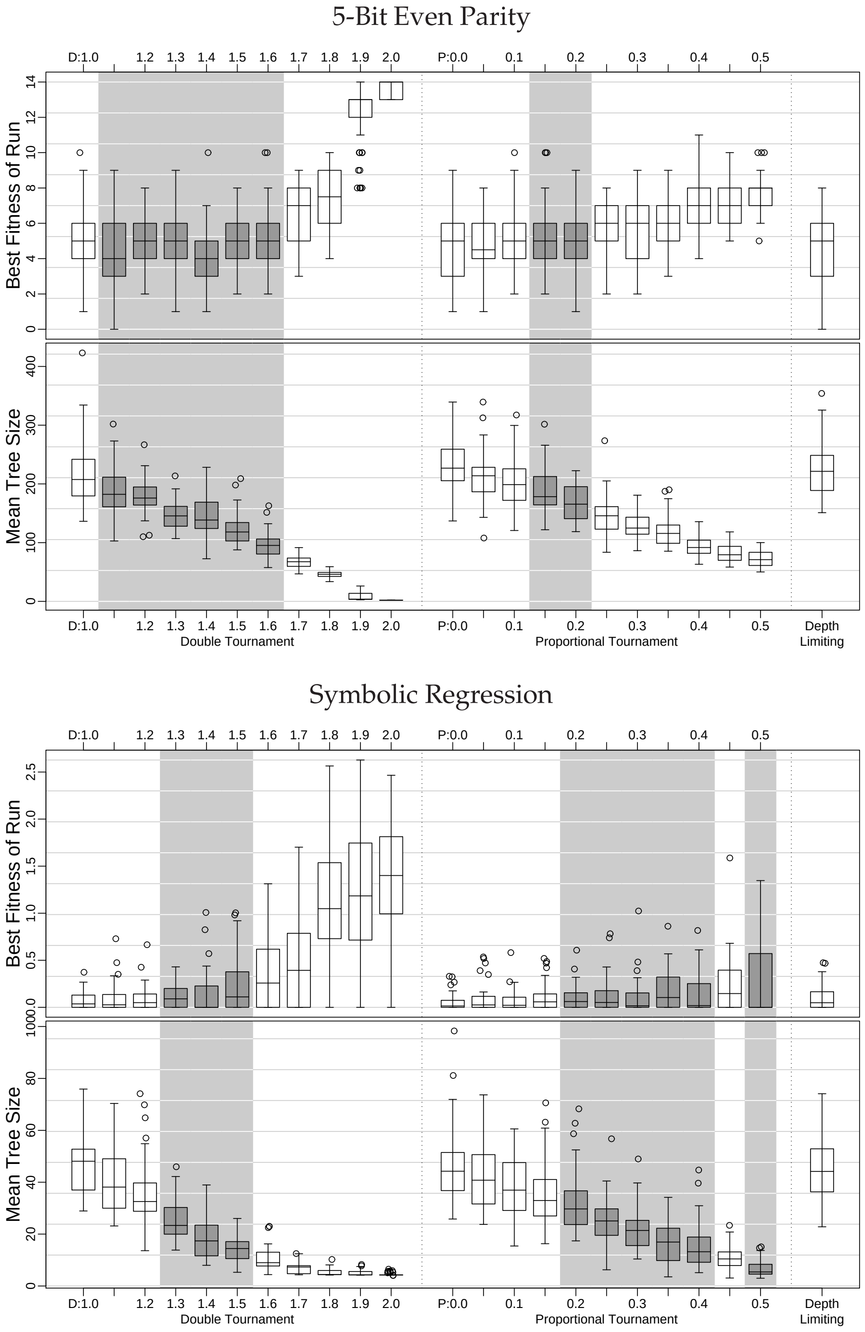
We fixed F to 7, a common value in genetic programming literature, and set *do-fitness-first* to true (in previous work (Luke and Panait, 2002a) we found that the setting of *do-fitness-first* made little difference). Thus the only parameter of consequence was D. Our initial experiments revealed that even D values as small as 2 put too much pressure on parsimony, and the fitnesses of the resulting individuals were statistically



Artificial Ant

11-Bit Boolean Multiplexer

Figure 7: Boxplot of best fitness and mean tree size of run for Double Tournament and Proportional Tournament methods (in combination with depth-limiting), as compared to Koza-style depth limiting alone. The size of the tournament on size for Double Tournament is denoted D on the graphs; the proportion of tournament based on size for Proportional Tournament is denoted P. The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.


5-Bit Even Parity

Symbolic Regression

Figure 8: Boxplot of best fitness and mean tree size of run for Double Tournament and Proportional Tournament methods (in combination with depth-limiting), as compared to Koza-style depth limiting alone. The size of the tournament on size for Double Tournament is denoted *D* on the graphs; the proportion of tournament based on size for Proportional Tournament is denoted *P*. The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

significantly worse than with no parsimony pressure at all. In order to rectify this matter, we permitted D to hold real values between 1.0 and 2.0, increasing by 0.1 each time. In this value range, two individuals participate to the tournament, and with probability $D/2$ the smaller individual wins, else the larger individual wins. Ties are broken at random. Thus $D = 1$ is random selection, while $D = 2$ is the same as a plain parsimony-based tournament selection of size 2.

Proportional Tournament This is an even easier technique. The Proportional Tournament algorithm selects an individual using tournament selection with size 7. However, a proportion of tournaments will select based on parsimony rather than on fitness. A fixed parameter P defines the proportion, where higher values of P imply less of an emphasis on fitness: $P = 0.0$ implies that all tournaments will select based on fitness, while $P = 0.5$ implies that tournaments will select on fitness or size with equal probability. We allowed P to range from 0 to 0.5 in increments of 0.05.

8.1 Experiment

The results for the two techniques are shown in Figures 7 and 8. We note that for some problems, such as Artificial Ant, the methods had wide swaths parameter settings which produced larger trees at equivalent fitness to depth limiting. Other problems, particularly 11-Bit Boolean Multiplexer, had very narrow settings.

As expected, increasing D and P leads to an increase in the parsimony pressure. This results in smaller tree sizes, but usually at the cost of slightly degrading the best fitness of run. Double Tournament has a wide range of good settings for Artificial Ant and 5-Bit Even Parity, but only a few such good settings in the other two problem domains. In particular, $D = 1.4$ was consistently better across all problems.

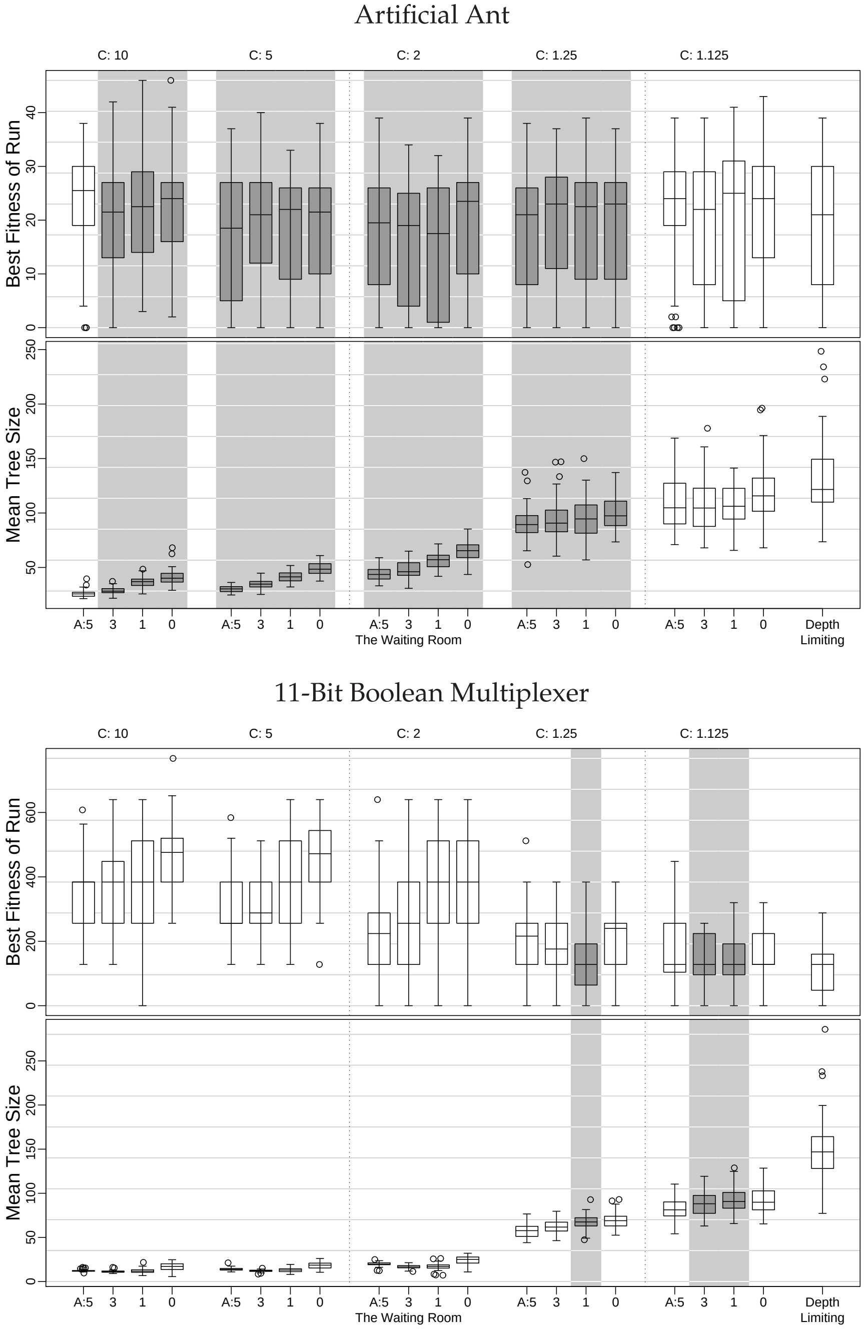
Similarly, Proportional Tournament has only two good settings for the 11-Bit Boolean Multiplexer and 5-Bit Even Parity domains, and a much larger set of such settings for Artificial Ant and Symbolic Regression. The setting $P = 0.2$ was also consistently better across all four problem domains.

Double Tournament and Proportional Tournament differ from one another in an important and intuitive fashion: an individual passes the Double Tournament if it is generally low in size *and* high in fitness; but passes the Proportional Tournament if it is generally low in size *or* high in fitness. We had expected this distinction to present itself in a significant difference in results, but the differences are actually relatively minor. Although it appears from the figures that Double Tournament more rapidly worsens in fitness, this is largely a matter of scaling: our presented Proportional Tournament results only went to $P = 0.5$. The two techniques follow very similar curves.

9 The Waiting Room

The Waiting Room is different from previously-discussed methods in that it does not consider size as part of the fitness-selection procedure, but rather punishes for code growth elsewhere in the evolutionary mechanism. Specifically, The Waiting Room adds a pre-birth phase to all newly created individuals (the “waiting room”). Children must wait in the waiting room for a certain period of time — larger children must wait longer — before they are permitted to enter the population and compete.

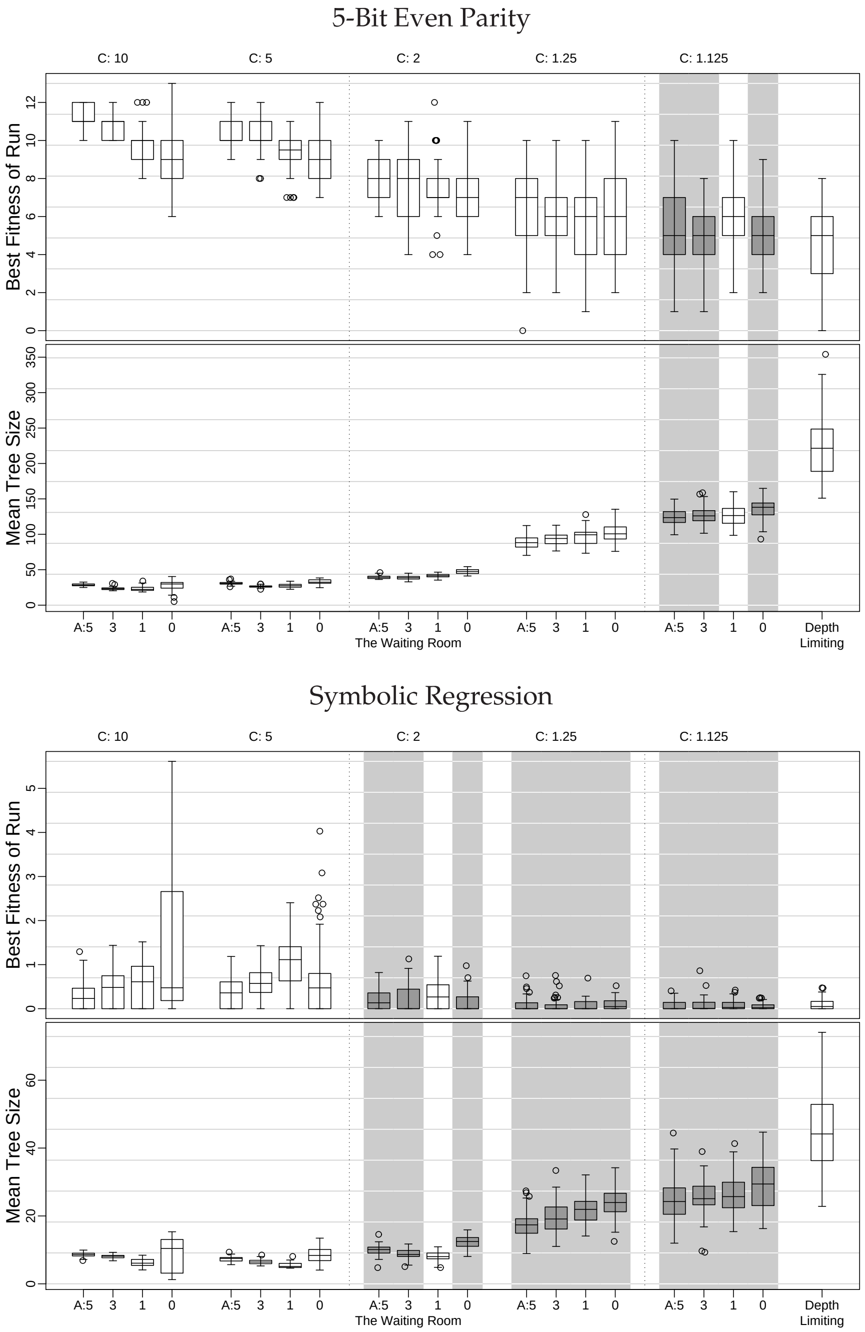
Let the population size be N. At each generation, some CN newly-created individuals are evaluated, then added to the waiting room. $C > 1$, so the waiting room will be larger than the final population size. Each individual in the waiting room is assigned a *queue value* equal to the individual’s size. Next, the N children with the



Artificial Ant

11-Bit Boolean Multiplexer

Figure 9: Boxplot of best fitness and mean tree size of run for The Waiting Room (in combination with depth-limiting), as compared to Koza-style depth limiting alone. The Waiting Room has two parameters: the ratio of child pool size to population size (denoted *C*), and the "Aging" parameter (denoted *A*). The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.



5-Bit Even Parity

Symbolic Regression

Figure 10: Boxplot of best fitness and mean tree size of run for The Waiting Room method (in combination with depth-limiting), as compared to Koza-style depth limiting alone. The Waiting Room has two parameters: the ratio of child pool size to population size (denoted *C*), and the “Aging” parameter (denoted *A*). The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

smallest queue values are removed from the waiting room and form the next generation. The remaining individuals have their queue values reduced by subtracting a value A. A counteracts queue stagnation: thus eventually even giant individuals may have a chance to be introduced to the population.

Our implementation of the waiting room allows it to grow without bound; but we imagine that real-world implementations would require some maximum size. Besides memory concerns, naive implementations of the waiting room impose significant computational complexity by cutting down individuals by A each time. We suggest using a binomial heap, and introducing new individuals to the queue by adding to them A times the generation number rather than subtracting A for the existing individuals in the heap. This imposes at most an $O(\lg |\text{Waiting Room}|)$ cost per selection.

The Waiting Room came about originally from the notion that larger children might take longer to be evaluated, and thus the evaluation process itself could provide a natural parsimony pressure by letting the fast, small children breed more rapidly through the population. Of course there are a great many problem domains for which evaluation time is not significantly affected by size; and introns would short-circuit such problem domains as well. We should mention a similarly-inspired method which we have *not* found to be effective: asynchronous island models. Island models connect parallel evolutionary processes via a network; every so often an evolutionary processes will "migrate" individuals to other processes over the network. In an asynchronous island model the processes are not synchronized by generation, but may run at their own pace. In theory processes with smaller individuals should run faster, and hence migrate more individuals to other processes, and those migrants will tend to be small. This gives small individuals a chance to spread throughout the network more rapidly. While we have not been able to get an asynchronous island model on its own to produce sufficiently parsimonious results, nonetheless we find the notion intriguing enough to be worth mentioning.

9.1 Experiment

The Waiting Room requires two parameters (C and A). We let C range from 1.125 to 10, doubling each time, and let $A = 0$, 1, 3, and 5. In most cases, these parameters were relatively independent. *Smaller* values of C and A each resulted in stronger parsimony.

The results are shown in Figures 9 and 10. We note that success is almost entirely dependent on the value of C, but effective settings for C are strongly dependent on the problem domain. Artificial Ant and 11-Bit Boolean Multiplexer (or 5-Bit Even Parity) are almost disjoint in their preferred settings for C. $C = 1.25$ and 1.125 results in improved results in the Artificial Ant and Symbolic Regression domains, independent of the aging parameter A. On the other hand, only a few settings (usually $C = 1.125$ and various values for A) result in similar performance in the 11-Bit Boolean Multiplexer and 5-Bit Even Parity domains. Overall, there were no settings for C and A which were consistently superior to depth limiting across all four problem domains. Except for Artificial Ant, $C = 1.125, A = 3$ was as good value, as was $C = 1.25, A = 1$ for all but 5-Bit Even Parity.

We note that the two domains where C had to be large were the boolean problems with large tree solutions, whereas Artificial Ant and Symbolic Regression both require small C values. At present we have no explanation for this: but this strong domain sensitivity suggests that there may be a significant difficulty in tuning this method for other problems as well.

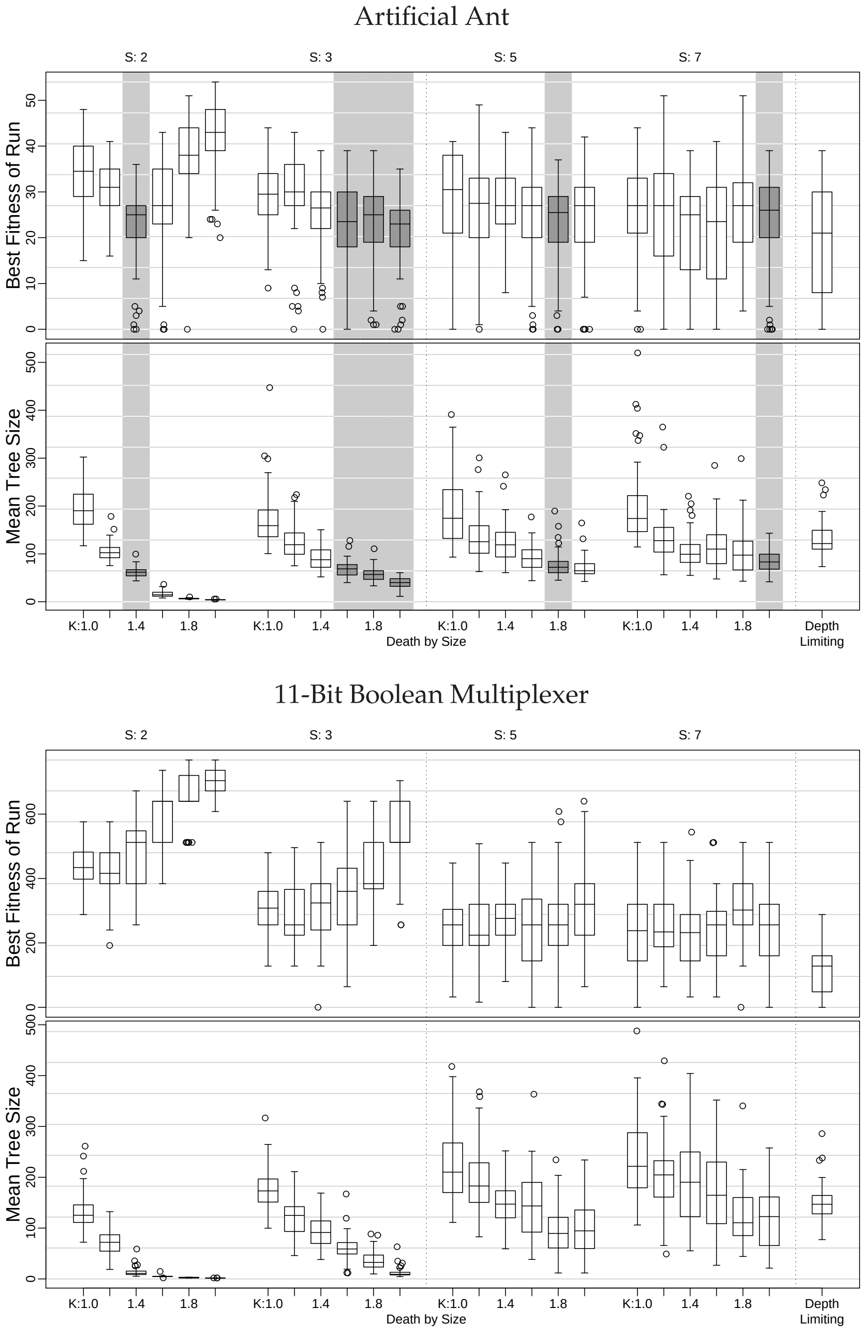
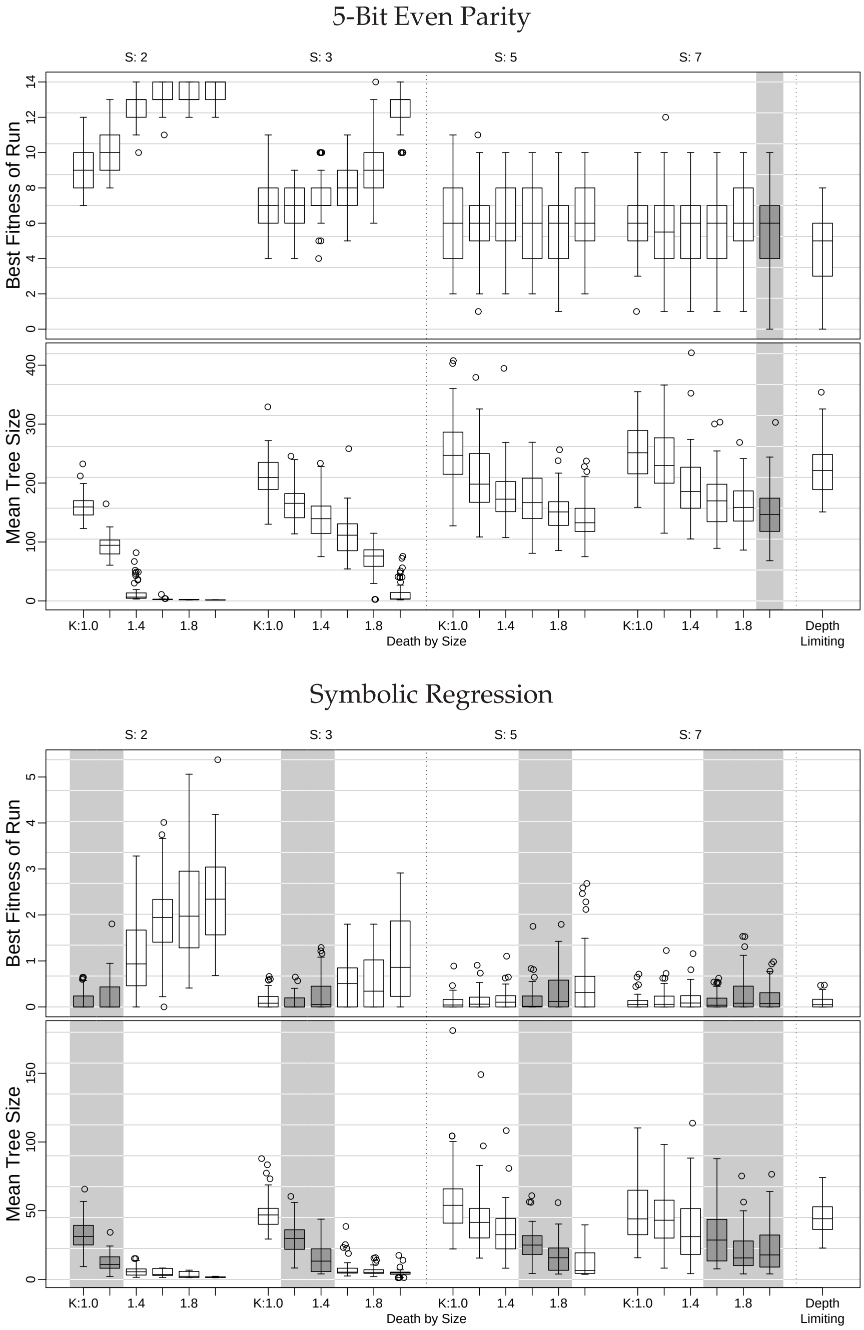


Figure 11: Boxplot of best fitness and mean tree size of run for Death by Size method (in combination with depth-limiting), as compared to Koza-style depth limiting alone. Death by Size has two parameters: the size of the selection tournament for breeding (denoted *S*), and the size of the selection tournament for individuals to "die" (denoted *K*). The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.



5-Bit Even Parity

Symbolic Regression

Figure 12: Boxplot of best fitness and mean tree size of run for Death by Size method (in combination with depth-limiting), as compared to Koza-style depth limiting alone. Death by Size has two parameters: the size of the selection tournament for breeding (denoted *S*), and the size of the selection tournament for individuals to "die" (denoted *K*). The settings on gray background have better or similar fitness (with 95% confidence), and significantly better tree size (with 99.995% confidence). On all graphs, lower values are better.

10 Death by Size

Like The Waiting Room, *Death by Size* is unusual in that it does not punish for size in the selection process. Instead it punishes for size in the *deselection* process. All methods presented so far can be applied to evolutionary computation methods where an entire population is created at each generation. But the *Death by Size* method presumes usage of a steady-state evolutionary algorithm, where at each time step some number of individuals are selected to breed children, while other individuals are selected to *die* and be replaced by the new children.

In Death by Size, selecting individuals to breed is done using fitness as the objective. But selecting individuals to die and be replaced is done by size: larger individuals are more likely to be selected for death. Tournament selection was used for both of these selection procedures. The size of the fitness-selection tournament was a parameter S, and the size of the size-selection tournament was a parameter K. As was the case for Double Tournament, we found that settings of K had to be very moderate, so we permitted K to range from 1.0 to 2.0. In this range, two individuals participate in a tournament, and with probability $K/2$ the larger individual wins, else the smaller individual wins. Ties are broken at random. As K increases, larger individuals are selected to die, resulting in stronger bloat control.

Because Death by Size is a steady-state method, it is in some sense odd to be comparing it (augmented with depth limiting) against *generational* plain depth limiting. We do so only for consistency. To compare otherwise, note that by setting $K = 1.0$, the death by size effect is eliminated and the problem reduces to plain depth limiting in a steady-state setting.

10.1 Experiment

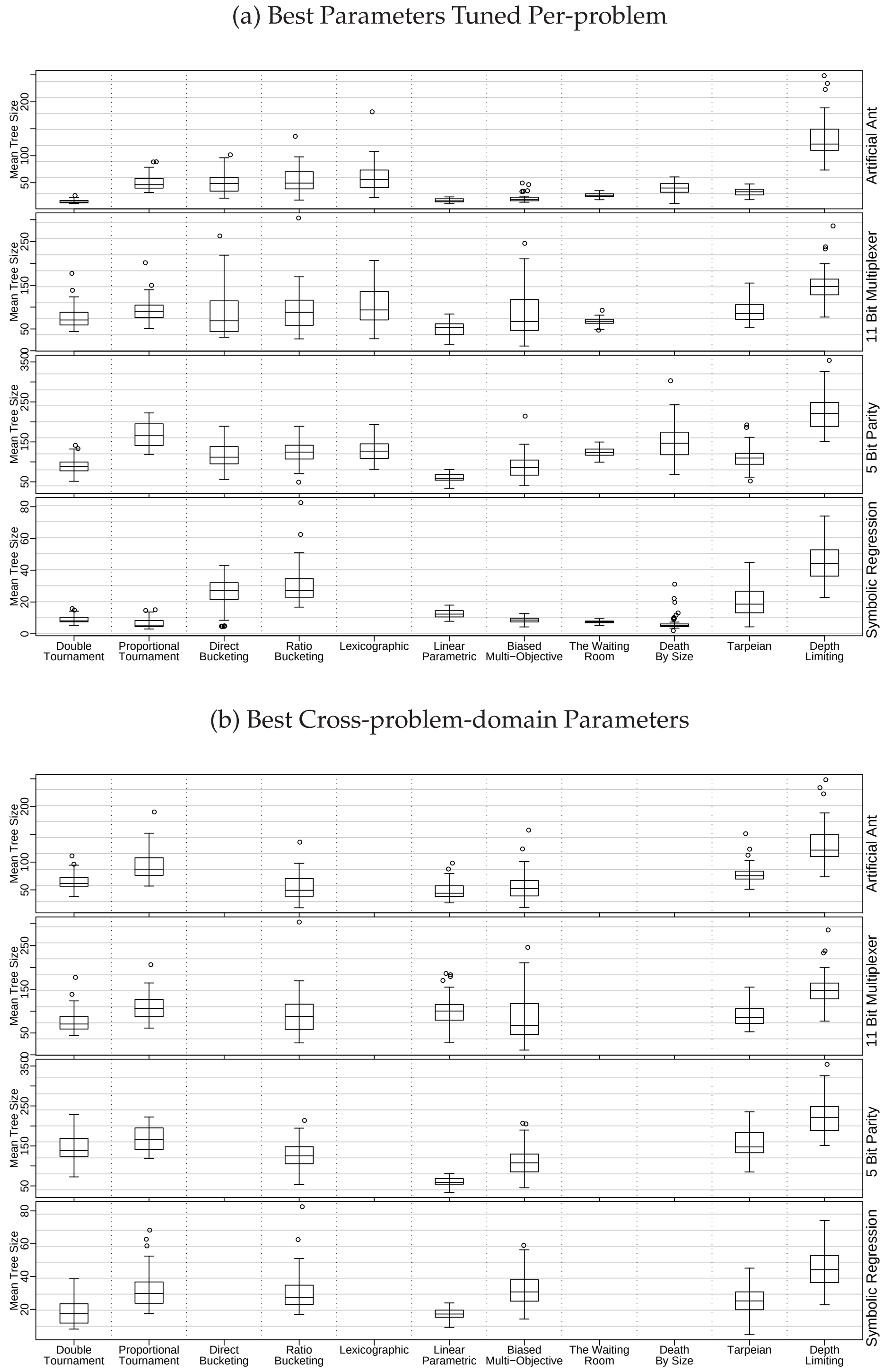
We allowed $S = 2$, 3, 5, or 7, and let K range from 1.0 to 2.0 in increments of 0.2. The results were quite surprising: under *no combination of settings* could Death by Size outperform plain depth limiting in the 11-Bit Boolean Multiplexer problem. In the 5-Bit Even Parity problem, Death by Size outperformed plain depth limiting only for a single setting ($S = 7, K = 2.0$)! We also observed an interesting relationship between the settings of S and K: in the Artificial Ant and the Symbolic Regression domains, good results are obtained when S and K are both correlated (either both small, both medium, or both large). Beyond the fact that 5-Bit Even Parity and 11-Bit Boolean Multiplexer were again our two large-solution boolean problems, we do not yet have an explanation for this trend.

We believe this is a symptom of the steady-state evolution procedure, which tends to apply a strong selection pressure and which we found to produce very large trees very rapidly. Indeed, we also performed experiments where Death by Size was *not* augmented with depth limiting, and this usually resulted in trees so large that they exceeded the memory capacity of our evolutionary computation system.

Nonetheless we retain Death by Size here because in Symbolic Regression it produced exceedingly small trees at equivalent fitness to plain depth limiting alone.

11 Comparison

Which method was most effective? This is a difficult question to answer. Some methods were quite good for certain problem domains, but poor, or in fact completely ineffective, for other problem domains. Other methods were more consistent in their success across domains.


(a) Best Parameters Tuned Per-problem

(b) Best Cross-problem-domain Parameters

Figure 13: Boxplot of mean tree size of run for the better settings of each of the methods, where the settings are chosen on a per-problem basis, or the best chosen from among successful cross-problem-domain settings.

Method	Ant	Multiplexer	Parity	Regression
Double Tournament (D, *do-fitness-first?*)	2.0, $true$	1.4, $false$	1.6, $true$	1.6, $true$
Proportional Tournament (P)	0.45	0.25	0.2	0.5
Direct Bucketing (B)	100	250	250	25
Ratio Bucketing (R)	1/2	1/2	1/6	1/2
Plain Lexicographic	(none)	(none)	(none)	(none)
Linear Parametric (X)	2	2	32	16
Biased Multi-objective (F)	0.8	0.95	0.9	0.75
The Waiting Room (A, C)	4, 10	1, 1.25	5, 1.125	2, 2.5
Death by Size (K, S)	2.0, 3	(none)	2.0, 7	1.7, 3
Tarpeian (W)	0.6	0.3	0.5	0.4

Table 1: Settings Used for Per-Problem Comparisons

We think the primary question an experimenter would ask is: lacking further heuristic information about the chosen problem domain, which technique is likely to work well in general on it? This is not an easy question to answer, but we chose our four problem domains both to provide some spread over the space of GP problems and to be common enough to be relevant to much of the literature.

As it has a continuous fitness function, Symbolic Regression has the characteristic that small improvements can always be made through additions to the existing tree; this is why Lexicographic Parsimony Pressure fails on the method. Artificial Ant exhibits strong relationships between nodes throughout the tree: changes in the left part of the tree have a dramatic effect on the operation in the right part of the tree due to execution order. Multiplexer and Parity are similar to one another in that they both have large-sized solutions; but Multiplexer is generally the most difficult of the four problems, whereas Parity is among the easier ones.

If a technique is successful over all four problems, we felt it was a reasonable candidate. Our "cross-platform" comparison in Section 11.2 is motivated in this way. However for the interested reader we begin with a "per-problem" comparison which asks which methods performed best on each of our four problem domains when tuned for that particular problem.

11.1 Per-Problem Comparison

We begin with a per-problem comparison to see which methods worked best *if* one knew the optimal parameter setting for that technique given the problem. We do not provide this data to suggest that a certain method is a better choice for a given problems: within problem domains there are many parameters which may favor one method or another. Instead, we primarily include a per-problem comparison for completeness, and to highlight certain problem features — particularly in Symbolic Regression — which can have unexpected effects on the methods used.

For each combination of method and problem domain, we selected a unique setting that we considered best. This was accomplished by eliminating all settings that were worse than plain depth limiting alone in fitness (using pair-wise comparisons at 95% confidence), or worse than or equivalent to depth limiting in mean tree size of run (using pair-wise comparisons at 99.995% confidence). We viewed the remainder as the set of parameter settings which an experimenter would find acceptable: reducing tree size while maintaining reasonable fitness. From this set we then chose the setting which had the smallest mean tree size of run. We compared by tree size because in these ranges the fitness values did not differ by much. For Double Tournament, we considered results both with *do-fitness-first* set to true and set to false; although there was no statistically significant difference, in one case setting *do-fitness-first* to false was slightly better. The boxplots of these best settings are shown in Figure 13a. The settings are shown in Table 1.

As can be seen in Figure 13a, for Lexicographic Parsimony Pressure was not superior to plain Depth limiting in the Symbolic Regression domain. Similarly, we could not find any setting for Death by Size to outperform Depth limiting in the Multiplexer domain.

11-Bit Boolean Multiplexer and 5-Bit Even Parity These two domains proved to have similar results despite 11-Bit Boolean Multiplexer being significantly more difficult. We attribute this to the two problem domains both being boolean, discrete-fitness functions. In the 11-Bit Boolean Multiplexer domain, the Linear Parametric and Biased Multi-objective methods were not worse than any other method. Of them, Linear Parametric significantly outperformed all nine remaining methods. The second tier contained Double Tournament, Direct Bucketing, Ratio Bucketing and The Waiting Room, all significantly worse than only Linear Parametric.

Linear Parametric also significantly outperformed all other methods in the 5-Bit Even Parity domain. The next best methods (dominated by only Linear Parametric) were Double Tournament (better than eight other methods) and Biased Multi-objective (better than six other methods).

Symbolic Regression Results were completely different in the Symbolic Regression domain, where Double Tournament, Proportional Tournament, The Waiting Room and Death by Size were the only non-dominated methods. These were followed closely by Biased Multi-objective. Linear Parametric had significantly worse code growth than all five of these methods.

We are not surprised that Linear Parametric performed somewhat unimpressively in this problem, as it entails fitnesses which even at very small values are significantly different, and late in evolution size considerations will come to dominate due to the linear combination of size and fitness. Perhaps more interesting is the poor showing of Tarpeian and the even worse showing of the lexicographic methods. Lexicographic techniques only break ties when the fitness is identical, and this allows Symbolic Regression to increase tree size as long as the fitness is just *barely* improved. We do not have a similar explanation for Tarpeian's performance.

Artificial Ant Here, Double Tournament was significantly better than all other methods except for Linear Parametric. Biased Multi-objective was outperformed by Double Tournament but not significantly by Linear Parametric. The fourth-ranked method was The Waiting Room, which was significantly worse than Double Tournament, Linear Parametric and Biased Multi-objective, but it was also significantly better than all other seven methods.

Summary Overall, Double Tournament and Biased Multi-objective appeared to have the best performance across all problem domains when tuned to their optimal per-problem values. Linear Parametric, even though not dominated in three domains, performed poorly by comparison in the Symbolic Regression problem.

11.2 Cross-Problem Comparison

Perhaps more of interest to the experimenter is the question: which method will give me the best results with problem-independent settings? We again selected a "best" setting, this time one which performed well across all problem domains. We first eliminated all settings worse than plain depth limiting alone in fitness, or worse or equivalent to plain depth limiting in mean tree size of run, in *any of the four problem domains*. This considerably reduced our remaining settings, and it eliminated from the running several methods, either because a method had no settings superior to depth limiting in a specific domain (Lexicographic, Depth by Size, as mentioned earlier), or because no single setting existed which was superior to plain depth limiting across all problems (Direct Bucketing, The Waiting Room). From the valid settings, we chose the one with the lowest mean tree size of run averaged over all four problem domains.[3] The settings chosen were:

Method	Settings
Double Tournament	D=1.4, *do-fitness-first*=false
Proportional Tournament	P=0.2
Lexicographic with Direct Bucketing	(not contending)
Lexicographic with Ratio Bucketing	R=1/2
Plain Lexicographic	(not contending)
Linear Parametric	X=32
Biased Multi-objective	F=0.95
The Waiting Room	(not contending)
Death by Size	(not contending)
Tarpeian	W=0.3

With these *problem-independent settings*, Ratio Bucketing, Linear Parametric and Biased Multi-objective were the only non-dominated methods in the Artificial Ant domain, followed closely by Double Tournament. For Multiplexer, Double Tournament, Ratio Bucketing, Linear Parametric, Biased Multi-objective, and Tarpeian were the five non-dominated methods. Proportional Tournament was only dominated by the Double Tournament method.

Similarly to when using domain-dependent settings, Linear Parametric significantly outperformed all other methods in the 5-Bit Even Parity domain. The second tier of methods consisted of Ratio Bucketing and Biased Multi-objective, followed closely by Double Tournament and Tarpeian, which were only significantly worse than Linear Parametric and Biased Multi-objective.

As The Waiting Room did not have any valid problem-independent settings, the ranking changed in the Symbolic Regression domain: Double Tournament and Linear Parametric were significantly better than all other methods. Tarpeian was significantly worse than only Linear Parametric, while Proportional Tournament, Ratio Bucketing and Biased Multi-objective were worse than both Double Tournament and Linear Parametric.

The surprise: overall, Linear Parametric with domain-independent settings *was consistently non-dominated across all four problems.* Other good choices for bloat control included Double Tournament, Ratio Bucketing, Biased Multi-objective, and Tarpeian. We wish to reiterate that the combination of a method with depth limiting was nearly

[3] As it turned out, this second step is not very relevant. Only two methods trigger this step, and of those only a single method (Ratio Bucketing) would be affected by the choice of methodology.

universally superior to either the method alone or depth limiting alone. Thus while some methods outperformed one another, when combined with depth limiting none of them is necessarily a *bad* choice.

12 Conclusions

In this paper we discussed the problem of code bloat and why it occurs, then introduced a variety of parsimony pressure mechanisms to keep the phenomenon under control. These methods all punish the individual, in one way or another, for being large in size, noting relationships between the methods and certain unusual features. While none of the methods is formally justified by theory, empirical evidence suggests that they are all reasonably effective at reducing bloat, with a few problem-specific counterexamples.

Some of these methods by themselves cannot outperform the mild bloat control introduced with Koza-style depth limiting. But in our experiments, we discovered an interesting phenomenon: augmenting any of these methods with depth limiting *never hurt*. Thus why not use it? For this reason, we decided to use depth limiting as the baseline because of its universality in the literature, and compare it against the augmented methods. We then compared the methods to each other using only those parameters under which each improved on the baseline. We chose four different problem domains in which to test them.

Specific Methods We found that, once augmented with depth limiting, all of the methods discussed are effective across many of our problem domains, and several have settings which are effective regardless of problem. One probably cannot go wrong with the cross-problem settings for Double Tournament, Proportional Tournament, Ratio Bucketing, Biased Multi-objective Parsimony Pressure, or Tarpeian.

The various Lexicographic Parsimony Pressure methods performed less admirably than we would have liked. Plain Lexicographic Parsimony Pressure failed to reduce bloat in the Symbolic Regression problem, largely because Symbolic Regression trees can usually improve fitness a small if negligible amount by growing in size. The bucketed versions of the method fared better but still unimpressively on this problem; Ratio Bucketing barely made it into our recommendations above.

Death by Size — our one steady-state method — was surprising. While it failed terribly in 11-Bit Boolean Multiplexer, it performed quite impressively in Symbolic Regression. This variance is telling, leading us to wonder what the effect of steady state evolution might have on the other methods presented as well, an opportunity for future work.

We were also surprised to find Double Tournament usually outperforming Proportional Tournament, given how similar the methods are in philosophy and implementation.

What about Linear Parametric Parsimony Pressure? This method outperformed all others in the 5-Bit Even Parity problem, and was consistently good in all other problems. It performed best in the cross-problem comparison, though Double Tournament was a close second. Given our past rhetoric about possible difficulties in tuning this method, we were surprised to find a single setting which performed admirably across all four of our chosen. But we still believe that Linear Parametric Parsimony Pressure can be problematic. It is not difficult to modify the raw fitness assessment procedure for any of these problem domains such that there is no such setting, and such a modification is hardly contrived. Lacking any sense beforehand of what setting to use, Linear Parametric Parsimony Pressure seems to us to be an unnecessary gamble. We think that our choice of problem domains was serendipitous for Linear Parametric Parsimony Pressure. Nonetheless, we cannot debate that, properly tuned, the technique was very effective in a number of them.

Future Work We chose to augment all of the methods with a depth limit of size 17 because this is the standard in the literature: but adjusting this limit might improve or worsen things. This additional variable would be an interesting line of future work. Some papers have examined the effect of different depth limits (Gathercole and Ross, 1996; Langdon and Poli, 1997a), but these have done so in the context of problems designed to produce worse results as the depth limit *increases*, for theoretical analysis purposes. Other work has created dynamically changing depth limits Silva and Almeida (2003). But we are not aware of an empirical examination of the depth limit parameter at present.

One of the real challenges in dealing with bloat is that while there are theoretical models explaining its onset, the use of such models to develop bloat-control techniques is still problematic. Practically all the evidence in the literature for the methods described here has been empirical. Exactly one parsimony pressure method (Tarpeian) can lay some claim to theoretical motivation or inspiration, but not actual derivation (Poli, 2003). This theoretical exposition can also be equally used to argue for every method presented here, but it gives us hope that other formal justifications may soon become available. We think a formal examination of the problem is an excellent area for future work.

In the mean time, the comparisons detailed in this paper may prove useful to the evolutionary computation experimenter. Bloat is an increasingly important topic in evolutionary computation, and particularly in genetic programming. Experimental analysis such as in this paper can provide hints for countering it in real-world problems, and data in the comparison results may provide clues suggesting underlying causes yet to be discovered.

References

Angeline, P. J. (1996). Two self-adaptive crossover operators for genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, pages 89–110. MIT Press, Cambridge, MA, USA.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann.

Bassett, J. K. and De Jong, K. A. (2000). Evolving behaviors for cooperating agents. In *International Syposium on Methodologies for Intelligent Systems*, pages 157–165.

Belpaeme, T. (1999). Evolution of visual feature detectors. In Poli, R., Cagnoni, S., Voigt, H.-M., Fogarty, T., and Nordin, P., editors, *Late Breaking Papers at EvoISAP'99: the First European Workshop on Evolutionary Computation in Image Analysis and Signal Processing*, pages 1–10, Goteborg, Sweden.

Bleuler, S., Brack, M., Thiele, L., and Zitzler, E. (2001). Multiobjective genetic programming: reducing bloat using SPEA2. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 536–543, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea. IEEE Press.

Blickle, T. (1996). *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich.

Burke, D. S., De Jong, K. A., Grefenstette, J. J., Ramsey, C. L., and Wu, A. S. (1998). Putting more genetics into genetic algorithms. *Evolutionary Computation*, 6(4):387–410.

Cavaretta, M. J. and Chellapilla, K. (1999). Data mining using genetic programming: The implications of parsimony on generalization error. In Angeline, P. J., Michalewicz, Z., Schoenauer, M., Yao, X., and Zalzala, A., editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1330–1337, Mayflower Hotel, Washington D.C., USA. IEEE Press.

de Jong, E. D. and Pollack, J. B. (2003). Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines*, 4(3):211–233.

Ekart, A. and Nemeth, S. Z. (2001). Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73.

Gathercole, C. and Ross, P. (1996). An adverse interaction between crossover and restricted tree depth in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA. MIT Press.

Gustafson, S., Ekart, A., Burke, E., and Kendall, G. (2004). Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 5(3):271–290.

Haynes, T. (1998). Collective adaptation: The exchange of coding segments. *Evolutionary Computation*, 6(4):311–338.

Iba, H., de Garis, H., and Sato, T. (1994). Genetic programming using a minimum description length principle. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press.

Kalganova, T. and Miller, J. (1999). Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In Stoica, A., Keymeulen, D., and Lohn, J., editors, *Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware (EH'99)*, pages 54–63, Piscataway, NJ. IEEE.

Kennedy, C. J. and Giraud-Carrier, C. (1999). A depth controlling strategy for strongly typed evolutionary programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 879–885, Orlando, Florida, USA. Morgan Kaufmann.

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

Langdon, W. B. and Poli, R. (1997a). An analysis of the MAX problem in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 222–230, Stanford University, CA, USA. Morgan Kaufmann.

Langdon, W. B. and Poli, R. (1997b). Fitness causes bloat. In Chawdhry, P. K., Roy, R., and Pant, R. K., editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London.

Langdon, W. B. and Poli, R. (1998). Genetic programming bloat with dynamic fitness. In Banzhaf, W., Poli, R., Schoenauer, M., and Fogarty, T. C., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 96–112, Paris. Springer.

Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1999). The evolution of size and shape. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, pages 163–190. MIT Press, Cambridge, MA, USA.

Lucas, S. (1994). Structuring chromosomes for context-free grammar evolution. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 130–135. IEEE.

Luke, S. (2000). *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD thesis, Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA.

Luke, S. (2003). Modification point depth and genome growth in genetic programming. *Evolutionary Computation*, 11(1):67–106.

Luke, S. (2005). ECJ 12: An EC and GP system in Java. http://cs.gmu.edu/∼eclab/projects/ecj/.

Luke, S. and Panait, L. (2002a). Fighting bloat with nonparametric parsimony pressure. In Guervos, J. J. M., editor, *Parallel Problem Solving from Nature (PPSN VII)*, pages 411–421. Springer.

Luke, S. and Panait, L. (2002b). Lexicographic parsimony pressure. In Langdon, W. B., Cantu-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., , and Jonoska, N., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836. Morgan Kaufmann.

Martin, P. and Poli, R. (2002). Crossover operators for a hardware implementation of GP using FPGAs and Handel-C. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 845–852, New York. Morgan Kaufmann Publishers.

Nedja, N., Abraham, A., and de Macedo Mourelle, L., editors (2006). *Genetic Systems Programming*. Springer.

Nordin, P. and Banzhaf, W. (1995). Complexity compression and evolution. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA. Morgan Kaufmann.

Nordin, P., Francone, F., and Banzhaf, W. (1996). Explicitly defined introns and destructive crossover in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, pages 111–134. MIT Press, Cambridge, MA, USA.

Olsson, R. (1995). Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81.

Poli, R. (2003). A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic Programming, Proceedings of EuroGP'2003*, pages 204–217. Springer.

Ryan, C. (1994). Pygmies and civil servants. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 11, pages 243–263. MIT Press.

Silva, S. and Almeida, J. (2003). Dynamic maximum tree depth. In *Genetic and Evolutionary Computation – GECCO-2003*, pages 1776–1787, Chicago. Springer.

Silva, S. and Costa, E. (2004). Dynamic limits for bloat control: Variations on size and depth. In *Genetic and Evolutionary Computation – GECCO-2004. Part 2*, pages 666–677. Springer.

Silva, S. and Costa, E. (2005). Resource-limited genetic programming: the dynamic approach. In Beyer, H.-G., O'Reilly, U.-M., Arnold, D. V., Banzhaf, W., Blum, C., Bonabeau, E. W., Cantu-Paz, E., Dasgupta, D., Deb, K., Foster, J. A., de Jong, E. D., Lipson, H., Llora, X., Mancoridis, S., Pelikan, M., Raidl, G. R., Soule, T., Tyrrell, A. M., Watson, J.-P., and Zitzler, E., editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1673–1680, Washington DC, USA. ACM Press.

Smith, P. W. H. and Harries, K. (1998). Code growth, explicitly defined introns, and alternative selection schemes. *Evolutionary Computation*, 6(4):339–360.

Smith, S. F. (1980). *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, Computer Science Department, University of Pittsburgh.

Soule, T. and Foster, J. A. (1998a). Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309.

Soule, T. and Foster, J. A. (1998b). Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA. IEEE Press.

Soule, T., Foster, J. A., and Dickinson, J. (1996). Code growth in genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA. MIT Press.

Tackett, W. A. (1994). *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems.

Wagner, N. and Michalewicz, Z. (2001). Genetic programming with efficient population control for financial time series prediction. In Goodman, E. D., editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 458–462, San Francisco, California, USA.

Wu, A., Schultz, A., and Agah, A. (1999). Evolving control for distributed micro air vehicles. In *IEEE Computational Intelligence in Robotics and Automation Engineers Conference*.

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