Genet Program Evolvable Mach (2009) 10:141–179
DOI 10.1007/s10710-008-9075-9

ORIGINAL PAPER

Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories

Sara Silva · Ernesto Costa

Received: 16 June 2008 / Revised: 5 December 2008 / Published online: 13 January 2009

Abstract Bloat is an excess of code growth without a corresponding improvement in fitness. This is a serious problem in Genetic Programming, often leading to the stagnation of the evolutionary process. Here we provide an extensive review of all the past and current theories regarding why bloat occurs. After more than 15 years of intense research, recent work is shedding new light on what may be the real reasons for the bloat phenomenon. We then introduce Dynamic Limits, our new approach to bloat control. It implements a dynamic limit that can be raised or lowered, depending on the best solution found so far, and can be applied either to the depth or size of the programs being evolved. Four problems were used as a benchmark to study the efficiency of Dynamic Limits. The quality of the results is highly dependent on the type of limit used: depth or size. The depth variants performed very well across the set of problems studied, achieving similar fitness to the baseline technique while using significantly smaller trees. Unlike many other methods available so far, Dynamic Limits does not require specific genetic operators, modifications in fitness evaluation or different selection schemes, nor does it add any parameters to the search process. Furthermore, its implementation is simple and its efficiency does not rely on the usage of a static upper limit. The results are discussed in the context of the newest bloat theory.

Keywords Genetic programming · Bloat · Dynamic limits · Review · Bloat theories

1 Introduction

Genetic Programming (GP) is the automated learning of computer programs [7, 28]. Theoretically, it can solve any problem whose candidate solutions can be measured

S. Silva (✉) · E. Costa
CISUC, University of Coimbra, Polo II – Pinhal de Marrocos, 3030-290 Coimbra, Portugal
e-mail: sara@dei.uc.pt

and compared, making it a widely applicable technique. Furthermore, the solutions found by GP are usually provided in a format that users can understand and modify to their needs. But its high versatility is also the cause of some difficulties. Users must set a number of parameters related to several aspects of the evolutionary process, some of which may influence the search process so strongly as to actually prevent an optimal solution to be found, if set incorrectly. And even when a reasonable match between problem and parameters is achieved, a major problem remains, one that has been studied for more than a decade: code growth.

The search space of GP is virtually unlimited and programs tend to grow in size during the evolutionary process. Code growth is a healthy result of genetic operators in search of better solutions, but it also permits the appearance of pieces of redundant code that increase the size of programs without improving their fitness. Besides consuming precious time in an already computationally intensive process, redundant code may start growing rapidly, a phenomenon known as *bloat*[1] [7, Chap. 7; 36, Chap. 11]. Bloat can be defined as an excess of code growth without a corresponding improvement in fitness. This is a serious problem in GP, often leading to the stagnation of the evolutionary process. Although many bloat control methods have been proposed (see [63, Chap. 2] for a review), a definitive solution is yet to be found.

This paper describes one of the newest approaches to bloat control. Unlike many others available, our approach does not require specific genetic operators, modifications in fitness evaluation or different selection schemes, nor does it add any parameters to the search process. It is inspired by the most traditional technique of imposing a fixed limit on the depth of the individuals allowed in the population, introduced by Koza in tree-based GP [28]. It implements a dynamic limit that can be raised or lowered, depending on the best solution found so far, and can be applied either to the depth or size of the programs being evolved. Depth limits can only be applied in the context of tree-based GP, but size limits are suitable also for linear GP [7], where size can be defined as the number of code lines. In tree-based GP, size can be defined as the number of tree nodes. The different variants of this approach will be collectively referred to as *Dynamic Limits* [65, 66].

The next section deals with bloat, describing the main theories regarding why it occurs. Section 3 describes the Dynamic Limits in detail, while Sect. 4 describes our test problems and specifies the techniques and parameters used in our experiments, also introducing and explaining the plots that are later used to present the results. Section 6 reports the results of the comparisons among the different techniques, while Sect. 6 discusses them and presents some considerations on the usage of limit restrictions in GP. Finally, Sect. 7 concludes and Sect. 8 provides ideas for future work.

2 Bloat

When Koza published the first book on GP [28], most of the evolved programs therein contained pieces of code that did not contribute to the solution and could be

[1] Or, as Bill Langdon put it, the “survival of the fattest”.

removed without altering the results produced. Besides imposing a depth limit to the trees created by crossover to prevent spending computer resources on extremely large programs, Koza also routinely edited the solutions provided at the end of each run to simplify some expressions while removing the redundant code.

Two years later, Angeline remarked on the ubiquity of these redundant code segments and, based on a slight biological similarity, called them *introns* [4]. In spite of classifying them as extraneous, unnecessary and superfluous, Angeline noted that they provided crossover with syntactically redundant constructions where splitting could be performed without altering the semantics of the swapped subtrees. Referring to some studies where the introduction of artificial introns was helpful or even essential to the success of genetic algorithms, Angeline revels in the fact that introns emerge naturally from the dynamics of GP. He even goes as far as to state that “it is important then to not impede this emergent property as it may be crucial to the successful development of genetic programs” [4].

It is possible that introns may provide some benefits. A non-intuitive effect that introns may have in GP is code compression and parsimony. It is not the bloated code full of redundant segments that is parsimonious, but the effective code that remains after removing the introns. Under specific conditions, particularly in the presence of destructive crossover, there is evidence that the existence of introns in the population results in shorter and less complex effective solutions [49, 72, 73]. Compact solutions are thought to be more robust and generalize better [27, 49, 61, 77, 81]. Introns also do seem to provide some protection against the destructive effects of crossover and other genetic operators [1, 11, 49, 70, 72] although this may not always be helpful. The usage of explicitly defined artificial introns has yielded generally good results in linear GP [38, 50, 51], but in tree-based GP it usually degraded the performance of the search process [3, 10, 70].

Regardless of its possible benefits to GP, the side effects of intron proliferation are very serious. Computational resources may be totally exhausted in the storage, evaluation and swapping of code that contributes nothing to the final solution, preventing GP from performing the effective search needed to find better solutions. Bloat is now widely recognized as a pernicious phenomenon that plagues most progressive search techniques based on discrete variable-length representations and using fixed evaluation functions [9, 30, 33, 35, 37]. Bloat control has become a very active research area in GP, already subject to different theoretic and analytic studies [32, 38, 48, 53, 54, 61, 62]. Several theories concerning why bloat occurs have been advanced, and many different bloat control methods have been proposed.

Next we describe the six main theories concerning the reasons why bloat occurs, along with some related ideas that are presented alongside the main theories. The different explanations for code growth are not necessarily contradictory. Some appear to be generalizations or refinements of others, and several most certainly complement each other. They are presented in logical, rather than precise chronological, order.

2.1 Hitchhiking

One of the first explanations for the multiplication of introns among GP programs, advanced by Tackett, was the *hitchhiking* phenomenon [77]. This is a common and undesirable occurrence in genetic algorithms, where unfit building blocks propagate throughout the population simply because they happen to adjoin highly fit building blocks. The introduction of artificial introns in genetic algorithms was partly an attempt to counteract the deleterious effects of hitchhiking.

According to the hitchhiking explanation, the reason why naturally emerging introns in GP become so abundant is that they, too, are hitchhikers. Tackett refutes the hypothetical protection against crossover (see Sect. 2.2) as the explanation for intron multiplication, based on the fact that the usage of *brood recombination* [1], a less destructive recombination strategy, did not result in less code growth [77]. An additional hypothesis for code growth, advanced by Altenberg and somewhat related to the *removal bias* theory later advanced by Soule (Sect. 2.3), suggested that it was caused by an “asymmetry between addition and deletion of code at the lower boundary of program size”, inherent to the recombination operator, and not dependent on selection pressure [2]. Tackett also refutes this hypothesis by showing that, on the contrary, code growth is directly proportional to selection pressure, and the only time bloat does not occur is when fitness is totally disregarded along the search process [77]. These results have later been reinforced by other experiments showing the absence of bloat when selection is random [8, 33, 38].

2.2 Defense against crossover

Although early disputed, the idea of *defense against crossover* as being the explanation for bloat has persisted in the literature for a long time [1, 11, 46, 49, 70, 72], also stated and referred to as the *replication accuracy theory* [46, 54], *intron theory* [21, 22, 76], and *protection theory* [12]. It is based on the fact that standard crossover is usually very destructive [7, Chap. 6; 49–51]. In face of a genetic operator that seldom creates offspring better than their parents, particularly in more advanced stages of the evolution, the advantage belongs to the individuals that at least have the same fitness as their parents, those who were created by neutral variations. Introns provide standard crossover and other genetic operators with genetic material where swapping can be performed without harming the effective code.

Curiously, most of the theory devoted to the defense against crossover was developed in the context of linear GP [49] and may not be completely applicable to tree-based GP [29, 40, 42, 61]. More specifically, introns can be roughly divided in two categories: inviable code and unoptimized code (or syntactic/structural and semantic introns [6, 12]). The former is code that cannot contribute to the fitness no matter how many changes it suffers, either because it is never executed or because its return value is ignored. The latter is viable code containing redundant elements whose removal would not change the return value [42]. Defense against crossover does not differentiate both types of introns, which is fine when considering only linear GP. But in tree-based GP the effects of regular genetic operators are very different in each type of intron. While inviable code effectively protects the individual from having its fitness changed, unoptimized code is highly susceptible to variations of its structure and its return value may greatly influence the fitness of the individual. It is not surprising to verify that the experiments supporting the defense against crossover in tree-based GP do not take into consideration any other type of intron besides inviable code.

Some of these experiments were performed by Soule and Foster, using a form of non-destructive *hill-climbing crossover* [52, 73] and studying its effects on code growth. In this crossover the offspring are kept only if they are strictly better than their parents in terms of fitness. Specifics apart, when offspring do not rise to these standards they are replaced by their parents. This crossover resulted in a strong limitation of code growth when compared to standard tree crossover, thus supporting the defense theory, but Luke suggests that code growth is just being delayed by the large amount of parents replicated along the generations [42, 43]. Additional experiments by Soule and Heckendorn using single node mutations have however suggested that code growth does occur in response to destructive operators [74].

Luke indeed rejects the defense theory in the context of tree-based GP [42] by using a simple procedure called *marking* [11]. Inviable code is identified and marked so that individuals cannot perform crossover within the inviable regions, thus removing the hypothetical advantage conferred by intron multiplication. The results showed a significant reduction of inviable code, but unoptimized code caused tree growth to persist and even increase. The defense theory seems to be correct when applied to those "syntactically redundant constructions" that Angeline called introns, but clearly does not apply to unoptimized code in tree-based GP. And even in linear GP, Brameier and Banzhaf have recently identified neutral crossover, not destructive crossover, as the main cause of code growth [12].

2.3 Removal bias

Although presenting evidence to support the theory of defense against crossover (Sect. 2.2), Soule performed additional experiments with another non destructive but less "rigorous" hill-climbing crossover [71, 72]. While the previous crossover retained only the offspring that were strictly better than their parents [73], this one retains all the offspring that are equal or better in terms of fitness. Both are non-destructive operators and yet the less rigorous one produces a substantial amount of code growth, although smaller than with standard crossover. Soule concludes that there must be a second cause for code growth besides the defense against crossover, and presents a theory called *removal bias* [37, 71, 72].

Given the general destructive nature of standard crossover, offspring having the same fitness as their parents often benefit from a selective advantage over their siblings. The presence of inviable code provides regions where removal or addition of genetic material does not modify the fitness of the individual. According to the removal bias, to maintain fitness the removed branches must be contained within the inviable region, meaning they cannot be deeper than the inviable subtree. On the other hand, the addition of a branch inside an inviable region cannot affect fitness regardless of how deep the new branch is. This asymmetry can explain code growth, even in the absence of destructive genetic operators. A related explanation had already been advanced by Altenberg (see Sect. 2.1).

When using the more rigorous non-destructive crossover that only allows offspring with better fitness than their parents (Sect. 2.2), removal bias is disabled and code growth effectively drops to a minimum, lending support to the theory. However, Luke suggests that the more rigorous crossover is probably causing an even larger amount of parent replication than the less rigorous crossover. He holds the argument that this may be stalling the evolution, which could be the only reason for the suppression of bloat [42, 43]. When using the more rigorous crossover, the improvement of mean population fitness is indeed slower than when using the less rigorous crossover [71]. When comparing only these two genetic operators, the effect of slowing down fitness improvement *and* code growth does suggest that the search process is simply being delayed, in this case by excessive parent replication. However, when compared with standard crossover, the more rigorous crossover improves fitness *faster*, despite producing much less code growth [71], which means the search process is not being hampered by parent replication. Soule and Heckendorn provided additional support to the removal bias theory by showing that crossover destructiveness is positively correlated with removed branch size, but mostly unaffected by inserted branch size [74].

2.4 Fitness causes bloat

The first theory that does not make introns responsible for bloat was advanced by Langdon and Poli [30, 33, 35, 37]. Also called *solution distribution* [72], *diffusion theory* [39, 40, 76], *drift* [12, 74], *nature of search spaces* [54] and *entropy random walk* [38], it has recently been identified simply by its main claim, *fitness causes bloat* [45]. Given its general characteristics, this theory is applicable to any progressive search technique using a discrete variable-length representation and a static evaluation function.

The fitness causes bloat theory basically states that with a variable-length representation there are many different ways to represent the same program, long and short, and a static evaluation function will attribute the same fitness to all, as long as their behavior is the same. Given the inherent destructiveness of crossover, when better solutions become hard to find there is a selection bias towards programs that have the same fitness as their parents. Because there are many more longer ways to represent a program than shorter ways, a natural drift towards longer solutions occurs, causing bloat. Although this explanation does not directly implicate introns in the process, the odds are that the code growth observed in the progressively longer alternative representations is ultimately caused by introns, either inviable or unoptimized code. Fitness causes bloat is strongly supported by theoretical evidence [36, Chap. 8].

If selection did not punish individuals worse than their parents, there would be no need to search for alternative representations for the same solutions, and bloat would not occur. So, fitness causes bloat. Confirming previous results by Tackett [77], experiments have shown that code growth does not occur when using random selection [8, 33, 38], not even when standard mutation is the only genetic operator [35]. Selection pressure has been further linked to code growth by Gustafson et al., who have found that increased problem difficulty induces higher selection pressure and loss of diversity, which together lead to bloat [23]. Studying bloat from a statistical learning theory viewpoint, Zhang and Mühlenbein have stated that programs tend to grow until they fit the fitness data perfectly [81], and Gelly et al., have also found evidence to support the claim that fitness causes bloat [21, 22].

2.5 Modification point depth

Another explanation for bloat in tree-based GP was advanced by Luke [39, 40, 42]. It has been called *depth-correlation* theory [76], but can also be referred to as *depth-based* theory or simply *modification point depth* [42].

Confirming previous results [24], Luke has observed that when a genetic operator modifies a parent to create an offspring, there is a correlation between the depth of the modified node and its effect on the fitness of the offspring when compared to the parent: the deeper the modification point, the smaller the change in fitness. Once again, because of the destructive nature of crossover, small changes will eventually benefit from a selective advantage over large changes, so there is a preference for deeper modification points. The larger the individual, the deeper its nodes can be, so large parents have an advantage over small parents. Plus, the deeper the modification point, the smaller the branch that is removed, thus creating a removal bias (Sect. 2.3). This may be regarded as a generalization of the original removal bias theory [74].

Luke denies that introns cause bloat [39]. In fact, according to the theory of modification point depth, size is a consequence of fitness, and Luke adds that size itself is what allows the propagation of inviable code [40, 42, 45]. Streeter also suggests that code growth may be related to a measure of resilience, where resilience is directly related to tree size [76].

2.6 Crossover bias

The most recent theory concerning bloat is the *crossover bias* theory by Poli et al. [15, 16, 55, 56]. It explains code growth in tree-based GP by the effect that standard subtree crossover has on the distribution of tree sizes in the population. Whenever subtree crossover is applied, the amount of genetic material removed from the first parent is the exact same amount inserted in the second parent, and vice versa. The mean tree size remains unchanged. However, as the population undergoes repeated crossover operations, it approaches a particular distribution of tree sizes (a *Lagrange distribution of the second kind* [25, 26, 55]), where small individuals are much more frequent than the larger ones. For example, crossover generates a high amount of single-node individuals. Because very small individuals are generally unfit, selection tends to reject them in favor of the larger individuals, causing an increase in mean tree size. It is the proliferation of these small unfit individuals, perpetuated by crossover, that ultimately causes bloat. The theory also holds for the popular 10/90% crossover that uses a non-uniform selection of crossover nodes, preferring non-terminal nodes with 90% probability.

Strong theoretical and empirical evidence supports the crossover bias theory. It has been shown that the bias towards smaller individuals is more intense when the population mean tree size is low, and that the initial populations resembling the Lagrange distribution bloat more easily than the ones initialized with traditional methods [15]. A somewhat unexpected finding was that one common bloat control method, the usage of size limits, actually speeds code growth in the early stages of the run. The reason is that size limits promote the proliferation of the smaller individuals, thus biasing the population towards the Lagrange distribution [16]. Along with further theoretical developments, it has also been shown that smaller populations bloat more slowly [56], and that elitism reduces bloat [57, 58].

2.7 Discussion

Looking back at all the bloat explanations suggested so far, one cannot help but notice the one thing that all the theories have in common, the one thing that if removed would cause bloat to disappear, ironically the one thing that cannot be removed without rendering the whole process useless: the search for fitness.

Remove fitness from the hitchhiking theory, and redundant code no longer propagates because the building blocks to which it associates cease to be selectively advantageous. Remove fitness from the defense theory, and individuals no longer need protection from a crossover that ceases to be destructive. Remove fitness from the removal bias theory, and the bias to remove small branches disappears. Remove fitness from the fitness causes bloat theory, and the drift towards longer alternative solutions no longer occurs. Remove fitness from the modification point depth theory, and deeper individuals no longer hold any advantage. Remove fitness from the crossover bias theory, and selection no longer rejects the numerous small individuals created by crossover. In short, remove fitness from the search process and bloat vanishes.[2]

All this may sound as obvious as saying that if you climb a mountain high enough you will suffer from lack of oxygen. Altitude causes lack of oxygen, so to avoid it you must not climb. But the goal *is* to climb! And yet the question remains: what causes the lack of oxygen, the climbing itself or the particular way you climb? Can we find a climbing technique to avoid lack of oxygen? Can we find the GP equivalent of the oxygen bottle?

3 Dynamic limits

This section describes our set of original bloat control techniques collectively designated as Dynamic Limits, from the initial idea of applying a dynamic limit to the depth of evolving trees, called Dynamic Maximum Tree Depth [65], to the variants where the limit can be applied to either depth or size, and is allowed to increase or decrease during the run [66].

[2] There are two assumptions to this statement: (1) Selection must be random, not only in terms of fitness but also in terms of size. A selection scheme that always selects the larger individuals would cause bloat; (2) Genetic operators must not, by themselves, bias the population towards larger sizes. An operator that takes an individual and always doubles its size would cause bloat.

3.1 Dynamic maximum tree depth

Tree-based GP traditionally uses a depth limit to avoid excessive growth of its individuals. When an individual is created that violates this limit, one of its parents is chosen for the new generation instead [28]. This technique effectively avoids the growth of trees beyond a certain point, but it does nothing to control bloat until the limit is reached. The static nature of the limit may also prevent the optimal solution to be found for problems of unsuspected high complexity.

3.1.1 Dynamic depth limit

Dynamic Maximum Tree Depth [65] is a bloat control technique inspired by the traditional static limit. It also imposes a depth limit on the individuals accepted into the population, but this one is dynamic, meaning that it can be changed during the run. The dynamic limit is initially set with a low value, but at least as high as the maximum depth of the initial random trees. Any new individual who breaks this limit is rejected and replaced by one of its parents instead (as with the traditional static limit), unless it is the best individual found so far. In this case, the dynamic limit is raised to match the depth of the new best-of-run and allow it into the population. Figure 1 shows the pseudo code of this procedure. The result is a succession of limit risings, as the best solution becomes more accurate and more complex.

Dynamic Maximum Tree Depth does not necessarily replace the traditional depth limit: both dynamic and fixed limits can be used at the same time (not shown in Fig. 1). When this happens, the dynamic limit always lies somewhere between the initial tree depth and the fixed depth limit. The simplicity of Dynamic Maximum Tree Depth makes it easy to use with any set of parameters and/or coupled with other techniques for controlling bloat.

The dynamic limit may also be used for another purpose besides controlling bloat. In real world applications, one may not be interested or able to invest a large

```
for all newly created individuals

    depth_i = depth of individual
    fitness_i = fitness of individual

    if depth_i ≤ dynamic_limit
        accept individual

        if fitness_i > best_fitness
            best_fitness = fitness_i

    if depth_i > dynamic_limit and fitness_i > best_fitness
        accept individual

        best_fitness = fitness_i
        dynamic_limit = depth_i
```

Fig. 1 Pseudo code of the basic Dynamic Maximum Tree Depth procedure (with no static limit)

amount of time in achieving the best possible solution, particularly in approximation problems. Instead, one may consider a solution to be acceptable only if it is sufficiently simple to be understood, even if its accuracy is known to be worse than the accuracy of other more complex solutions. Plus, shorter solutions tend to generalize better (Sect. 2).

One way to avoid the over-specialization of the solutions found by GP is to choose termination criteria that will not force the evolutionary process to go on indefinitely in search of a perfect solution. A less stringent stop condition yields a somewhat inaccurate solution, but one that is also simpler and hopefully generalizes better. However, setting the right stop condition may be a major challenge in itself, as one cannot predict the complexity needed to achieve a certain level of accuracy. By starting the search with a low dynamic limit for tree depth, the search is forced to concentrate on simple solutions first. The limit is then raised when a new solution is found that is more complex, but also more accurate, than the previous one. As the evolution proceeds, the limit is repeatedly raised as more and more complex solutions achieve increasingly higher levels of accuracy. Regardless of the stop condition, the Dynamic Maximum Tree Depth technique can in fact provide a series of solutions of increasing complexity and accuracy, from which the users may choose the one most adequate to their needs.

3.1.2 *Early results*

Early tests have shown that Dynamic Maximum Tree Depth is able to effectively contain code growth in a Symbolic Regression and the Even-3 Parity problems [65]. Two different settings for the initial value of the dynamic limit were tried: 6 and 9. The first value (6) is the traditional maximum depth of the trees in the initial population. Setting the dynamic limit to this value means that trees cannot even grow beyond their initial maximum depth unless they prove to be the best so far. Using the second value (9), trees can grow freely from their initial maximum depth of 6 until they reach depth 9, and only then see their growth restricted by the dynamic limit. The most restrictive value (6) resulted in lower mean tree size along the run without any impairment on the ability to converge to good solutions. Dynamic Maximum Tree Depth was also tested against and together with another bloat control technique, Lexicographic Parsimony Pressure [44]. Lexicographic Parsimony Pressure is based on a modified tournament that always selects smaller trees when their fitness is the same. The experiments showed a clear superiority of the dynamic limit, with the best results achieved when both techniques were coupled together.

3.2 Variations on size and depth

The original Dynamic Maximum Tree Depth was soon extended to include additional variants: a *heavy* dynamic limit, called heavy because it falls back to lower values whenever allowed, and a dynamic limit on *size* instead of depth.

Figure 2 shows the general acceptance procedure (including all the variants, using no static limit) that all newly created individuals must pass before being

```
for all newly created individuals

    illegal_parents_i = whether individual has illegal parents

    if illegal_parents_i
       my_limit_i = size/depth of largest/deepest parent
    else
       my_limit_i = dynamic_limit

    size_i = size/depth of individual
    fitness_i = fitness of individual

    if size_i ≤ my_limit_i
       accept individual

       if fitness_i > best_fitness
          best_fitness = fitness_i

       if VeryHeavy
       or (Heavy and size_i ≥ initial_dynamic_limit)
          dynamic_limit = size_i

    if size_i > dynamic_limit and fitness_i > best_fitness
       accept individual

       best_fitness = fitness_i
       dynamic_limit = size_i
```

Fig. 2 Pseudo code of the general Dynamic Limits acceptance procedure (all variants, no static limit). This is an extension of the procedure in Fig. 1. Only the shaded code is completely new

accepted into the new generation. This is an extension of the procedure in Fig. 1. Only the shaded parts are completely new. Both the new code and the small differences in the common code will be explained in the next sections. Any individual that does not meet the size/depth/fitness requirements of the Dynamic Limits method will not be accepted by this procedure, but instead replaced by one of its parents.

3.2.1 Heavy dynamic limit

Dynamic Maximum Tree Depth is capable of withstanding a considerable amount of parsimony pressure, as proven by the results obtained by initializing the dynamic limit with the lowest possible value, the maximum depth of the initial random trees [65] (Sect. 3.1.2). So there seems to be no reason why the limit should not be allowed to fall back to lower values in case the depth of the new best individual becomes lower than the current limit, an occurrence which is actually very common. So the first variation introduced to the original Dynamic Maximum Tree Depth is the *Heavy* dynamic limit, one that accompanies the depth of the best individual, up or down, with the sole constraint of not going lower than its initialization value [66]. An additional variation is the *VeryHeavy* limit, similar to the heavy variant but allowed to fall back even below its initialization value. Both these variants are covered in the second shaded block of Fig. 2.

As expected, whenever the limit falls back to a lower value, some individuals already in the population immediately break the new limit, becoming ‘illegals’. There was a vast range of options to deal with them, the more drastic being their immediate removal from the population, possibly replacing them by new random individuals. However, since these new ‘illegals’ could be the ones who managed to produce the new best individual, eliminating them could be harmful for the search process. A much softer option was adopted: the ‘illegals’ are allowed to remain in the population as if they were not breaking the limit, but when breeding, their children cannot be deeper than the deepest parent. This naturally and gradually places the population within limits again. The first shaded block of Fig. 2 deals with choosing the right limit (`my_limit_i`) to use for the new individual, depending on whether it has illegal parents. The first comparison involving the variable `dynamic_limit` in Fig. 1 (`if size_i` ≤ `dynamic_limit`) is now performed using the variable `my_limit_i` in Fig. 2 (`if size_i` ≤ `my_limit_i`).

3.2.2 Dynamic size limit

Even though bloat is known to affect many other search processes using variable-length representations (Sect. 2), depth limits cannot be used on non tree-based GP systems. Extending the idea of a dynamic limit to other domains must begin with the removal of the concept of depth, replacing it with the concept of size. The second variation on the original Dynamic Maximum Tree Depth is the usage of a dynamic *size* limit, where size is the number of nodes [66]. If a static limit is to be used along with this dynamic limit, it should also be on size, not depth. The variable `depth_i` of the pseudo code in Fig. 1 is now called `size_i` in Fig. 2, although it can refer to either size or depth.

Tree initialization in tree-based GP also typically relies on the concept of depth. This is the case with the popular Grow, Full, and Ramped Half-and-Half initialization methods [28]. Both Grow and Full methods create trees by adding random nodes until a maximum specified depth. The Grow method adds internal or terminal nodes, except at the maximum depth, where the choice is restricted to terminals. This creates trees with different shapes and sizes. The Full method creates balanced trees, with all terminal nodes at the maximum depth. It does this by adding random internal nodes except at the maximum depth, where it selects only terminals. The Ramped Half-and-Half method is a combination of the two, where half the population is initialized with Grow, and the other half with Full. In each half trees are created with depth limits ranging from 2 to a specified maximum value, ensuring a very diverse initial population.

When using the dynamic size limit, it makes no sense to keep using depth as a restriction on tree initialization. So a modified version of the Ramped Half-and-Half initialization method was created [66], where an equal number of individuals are initialized with sizes ranging between 2 and the initial value of the dynamic size limit. For each size, half or the individuals are initialized with the Grow method, and the other half with the Full method, that have also been modified to fit the size constraints only. In the modified Grow method, the individual grows by addition of random nodes (internal or terminal) without exceeding the maximum specified size;

```
nodes_left = maximum specified size

while nodes_left > 0

   if nodes_left = 1
      pool = terminals

   else
      selected_functions = functions with arity < nodes_left

      if Grow
         pool = selected_functions + terminals

      if Full
         pool = selected_functions
         if pool is empty
            pool = terminals

   add a random node from the pool
   nodes_left = nodes_left − 1
```

Fig. 3 Pseudo code of the modified Grow and Full initialization methods

the modified Full method chooses only internal nodes until the size is close to the specified, and only then chooses terminals. Unlike the original Full method, it may not be able to create individuals with the exact size specified, but only close (and never exceeding). Figure 3 shows the pseudo code of both methods.

3.2.3 *Early results*

Both heavy and size variations have been tested on the same problems as the original Dynamic Maximum Tree Depth technique (Symbolic Regression and Even-3 Parity) against and coupled with Lexicographic Parsimony Pressure [66] (Sect. 3.1.2). The heavy dynamic limit adds parsimony pressure during the run. Even without taking drastic measures towards the individuals that suddenly break the lower limit, the mean tree size along the run was kept significantly lower than with either the original Dynamic Maximum Tree Depth or Lexicographic Parsimony Pressure alone. Once again the best results were achieved by joining both techniques, and fitness was still not affected by such high levels of parsimony pressure. The dynamic size, however, did not perform as well in one of the problems (Parity), where the ability to find good solutions was compromised. These early results did not yet include the VeryHeavy variation.

3.2.4 *New implementation of dynamic size*

Because the early results obtained with the dynamic size techniques were not so brilliant, a new implementation of the size limit was developed and tested for this paper. In the previous implementation, when the population approached the limit it became very difficult for the parents to produce valid offspring. Any slight increase of the number of nodes usually conflicted with the limit, causing the new offspring

```
if not illegal_parents_i
    accept individual
```

Fig. 4 Additional pseudo code for the implementation of the new dynamic size limit (size limits only, no static limit)

to be rejected. It was hypothesized that this total lack of freedom to explore the search space around the current solution was the cause for the poorer results of the dynamic size techniques.

With the new implementation it becomes possible to explore the search space beyond the current size limit. When breeding, if both parents are within the limit then their offspring will be accepted into the new generation, regardless of their size. But the children who break the limit are accepted as 'illegals', the same status given to the individuals that suddenly break the limit when it falls back in the heavy variants (Sect. 3.2.1). From then on, the same simple restriction applies to all the illegal individuals: when breeding, their children cannot be deeper than the deepest parent. Figure 4 shows the supplemental pseudo code (to be added at the end of the procedure in Fig. 2) for the implementation of the new dynamic size limit.

4 Experiments

This section introduces the experiments that were performed in order to study the efficiency of Dynamic Limits as a bloat control method. It provides a description of the problems used as benchmarks, summarizes the set of tested techniques, specifies the procedures and parameter settings used in the experiments, and finally describes how the results will be presented.

All the experiments were performed with tree-based GP in generational mode using GPLAB[3]—A Genetic Programming Toolbox for MATLAB[4] [64]. Statistical significance of the null hypothesis of no difference was determined with (pairwise) Kruskal–Wallis ANOVAs at $p = 0.01$. A non-parametric ANOVA was used because the data is not guaranteed to follow a normal distribution. For the same reason, the median was preferred over the mean in all the evolution plots (Sect. 4.4). The median is also more robust to outliers, and has already been used in similar studies [75].

4.1 Problems

Four different problems were chosen to test the Dynamic Limits: Symbolic Regression, Artificial Ant, 5-Bit Even Parity and 11-Bit Boolean Multiplexer. This particular set of problems was chosen because it has been widely used in the literature [13, 14, 17–19, 31, 32, 39–45, 59, 60, 63, 65–69, 78–80], as it represents a

[3] http://gplab.sourceforge.net.

[4] http://www.mathworks.com/matlab.

varied selection in terms of bloat dynamics and response to different bloat control techniques.

4.1.1 Symbolic regression

The goal of the Symbolic Regression problem is to evolve a function that best approximates a set of points. In this particular case, 21 equidistant points of the quartic polynomial ($x^4 + x^3 + x^2 + x$) in the interval -1 to $+1$ are used.

The function and terminal sets for this problem are, respectively, $\{+, -, \times, \div, sin, cos, log, exp\}$ and $\{x\}$ (no random constants are used). The division and logarithm are protected as in [28]: the division returns 1 whenever the denominator is 0, and the argument of the logarithm is always converted to its absolute value. Fitness is measured as the sum of the absolute differences between the expected and predicted values of each point. It can take any real non-negative number, so there is a potentially infinite number of possible fitness values. Early results [65, 66] have suggested that the Symbolic Regression problem is not prone to the propagation of inviable code, but very much affected by unoptimized code, although these notions are contradicted in [45]. For simplicity, from now on this problem will be referred to simply as Regression.

4.1.2 Artificial ant

In the Artificial Ant problem the goal is to evolve a strategy to follow a food trail. In this particular case, the Santa Fe trail is used. The trail is represented on a 32 × 32 (toroidal) grid and the ant begins its search on the upper left corner, facing east.

The function and terminal sets for the Artificial Ant problem are, respectively, {*if-food-ahead, progn2, progn3*} and {*left*, *right*, *move*}, as defined in [28]. With the *if-food-ahead* function, the ant checks the cell directly in front of it and performs a certain action in case it finds a food pellet there. *progn*2 and *progn*3 allow the ant to perform any two or three consecutive actions. With the terminals *left* and *right* the ant can turn around 90° without moving from its cell. *move* allows the ant to move to the adjacent cell it is facing. When the ant stands on a cell containing a food pellet, it immediately eats it. A foraging strategy is built using these functions and terminals and each ant is given 400 time steps[5] to apply it repeatedly in search of the 89 food pellets available in the trail. Fitness is measured as the number of pellets remaining afterwards. Many different foraging strategies may result in the same fitness, and this seems to be correlated to the proliferation of inviable code.

[5] The number of time steps actually used in the original work by Koza [28] was 600, but a typographical error caused the number 400 to become more popular in the literature, the reason why we also use it.

4.1.3 5-Bit even parity

The 5-Bit Even Parity problem is in fact a symbolic regression problem where the function to evolve takes five boolean arguments and returns a single output indicating the parity of the arguments: 1 (or true) if an even number of arguments are 1, and 0 (or false) otherwise.

This problem uses the function and terminal sets {*and*, *or*, *nand*, *nor*} and $\{x_1,...,x_5\}$, respectively. Fitness is measured as the number of misclassified cases, so it may only take values between 0 and 32, even fewer than in the Artificial Ant problem. Once again, a large amount of structurally distinct individuals may have the same fitness. For simplicity, from now on this problem will be referred to simply as Parity.

4.1.4 11-Bit Boolean multiplexer

Also similar to a symbolic regression problem, the 11-Bit Boolean Multiplexer problem can however be viewed as a problem of electronic circuit design. The function to evolve takes three address arguments (a_0, a_1, a_2) plus eight data arguments ($d_0,\ldots,d_7$), all boolean. The value returned by the function is the particular data bit that is singled out by the address bits.

The 11-Bit Boolean Multiplexer problem uses the function and terminal sets {*and*, *or*, *not*, *if*} and $\{a_0, a_1, a_2, d_0,...,d_7\}$, respectively. Note that both address and data arguments are simply treated as terminals, undistinguishable from one another. Fitness is measured as the number of misclassified cases. This theoretically allows fitness values between 0 and 2048, but in practical terms the values usually fall into multiples of 32 [45]. The 11-Bit Boolean Multiplexer problem suffers from relatively little inviable code [45]. For simplicity, from now on it will be referred to simply as Multiplexer.

4.2 Techniques

Table 1 summarizes all the techniques compared within the Dynamic Limits approach. Koza is the baseline technique because of its popularity, and also to stress the improvements introduced when a dynamic limit is used instead of a static one. The names (and acronyms) of the other techniques are composed of several parts to

Table 1 Techniques compared within the Dynamic Limits approach

Technique	Acronym	Short description
Koza	K	Static depth limit
DynDepth	D	Dynamic depth limit
DynNodes	N	Dynamic size limit
hDynDepth	hD	Heavy dynamic depth limit
hDynNodes	hN	Heavy dynamic size limit
vhDynDepth	vhD	Very heavy dynamic depth limit
vhDynNodes	vhN	Very heavy dynamic size limit

Table 2 Settings used in the experiments

Runs	30
Generations	50
Population size	1000
Population initialization	Ramped Half-and-Half
Selection for reproduction	Tournament size 7
Genetic operators	Tree crossover, no mutation
Reproduction rate	0.1
Selection for survival	No elitism

help their identification: *Dyn* stands for Dynamic Limits; *Depth* (*D*) and *Nodes* (*N*) relate to depth and size limits, respectively; *h* and *vh* identify the respective Heavy and VeryHeavy variants. All the dynamic size techniques follow the new implementation described in Sect. 3.2.4.

4.3 Settings

Table 2 lists the parameters common to all the experiments. A total of 30 runs were performed with each technique for each problem. All the runs used populations of 1000 individuals allowed to evolve for 50 generations. Most of the remaining parameters follow the settings indicated in [28] and [45]. The initial populations were generated with the Ramped Half-and-Half procedure [28], modified when using the dynamic size limit (see Sect. 4.3.2 for details). Although some effort was put into promoting the diversity of the initial population, the tree initialization procedure does not guarantee that all individuals are distinct from one another. For each newly created individual that is structurally identical to any of the members already in the population, the process is retried until a different individual is generated or until 20 attempts have been made.

In all four problems, fitness was calculated such that lower values represent better fitness. Selection for reproduction was made with tournaments of size 7. A reproduction rate of 0.1 was used, meaning that there was a 10% probability of copying an individual intact into the next generation instead of choosing a genetic operator to create new individuals. Standard tree crossover was used, but with uniform distribution of the random crossover points, instead of the more typical 10/90% choice of terminal/internal nodes. It has been suggested that a leaf crossover higher than 10% may be beneficial [5], and total random selection of crossover points may not even affect the results [40]. No mutation was used. Selection for survival was not elitist (in the traditional sense only, since the techniques using variable size populations can be considered highly elitist, and the reproduction rate is also a form of elitism), meaning that the best individual of a given generation is not guaranteed to survive into the next generation.

4.3.1 Depth limits

Table 3 specifies the maximum depth/size of the individuals on the initial population, as well as the minimum and maximum limit values, for all the techniques compared

Table 3 Limits used within the Dynamic Limits approach

Technique	Initial population	Minimum limit	Maximum limit
Koza	6	17	17
(h)DynDepth	6	6	–
(h)DynNodes	20/105/50[a]	20/105/50[a]	–
vhDynDepth	6	–	–
vhDynNodes	20/105/50[a]	–	–

[a] Regression/artificial ant/parity & multiplexer

within the Dynamic Limits approach. The limit of the Koza technique has the same minimum and maximum value, as it remains static along the run. Heavy and non-heavy variants use the same limit range (hence they appear together), with the only difference that the non-heavy techniques can only increase the limit, while the heavy techniques can also decrease it as low as the maximum depth/size allowed on the initial population. The limit of the very heavy variants has no lower bound. An upper bound exists only in the Koza technique, with the traditional value of 17. The maximum depth of the individuals on the initial population is the also traditional value of 6. Whenever a dynamic limit is used, its initial value is exactly the same as the maximum depth/size allowed on the initial population.

4.3.2 Size limits

In terms of size instead of depth, appropriate values had to be found that somehow produced the same behavior as their corresponding values for depth. Tree initialization was performed using the modified Ramped Half-and-Half procedure described in Sect. 3.2.2.

The characteristics of the initial random population depend on the tree initialization method and the maximum depth/size allowed for the new individuals. They also depend on the function and terminal sets, with functions of higher arity producing bushier trees, and a high frequency of terminals producing sparser trees. So it is no surprise to verify that, when performing a depth based initialization, the total amount of nodes of all the individuals in the new population is different for the several problems considered.

When switching from depth to size, we searched for size limits that would generate populations containing similar amounts of nodes as the ones observed for the depth based initializations. We began by calculating the median amount of nodes of 30 depth based initializations, using a maximum depth of 6. Several sets of 30 size based initializations were then attempted, using different values for the maximum tree size (in multiples of 5). We selected the size limit that resulted in a median amount of nodes closest to the median found for the corresponding depth based initializations. As expected, this value was different for each problem: 20 nodes for the Regression problem, 105 nodes for the Artificial Ant, and 50 nodes for the Parity and Multiplexer problems. These are listed in Table 3. There is no maximum size limit.

4.4 Plots

The results of the experiments will be presented as boxplots and evolution curves concerning several aspects of the evolutionary process. There will be three figures for each problem, each figure containing more than one plot.

The first figure is dedicated to fitness, and contains two plots. See, for example, Fig. 6. On the left (a) there is a boxplot of the best fitness achieved by each technique during the run. Each technique is represented by a box and pair of whiskers. Each box has lines at the lower quartile, median, and upper quartile values, and the whiskers mark the furthest value within 1.5 of the quartile ranges. Outliers are represented by $+$, and $\times$ marks the mean. Any comparative statement regarding the performance of the techniques appearing on this boxplot is supported by statistical evidence, as described in the beginning of this section. This boxplot also indicates the success rate achieved by each technique, on the bottom of each box (numbers in bold). The success rate is the percentage of runs that found an optimal solution. On the right (b) there is a plot showing the evolution of the best fitness along the run, one line per technique. The evolution curves are obtained by connecting the 51 values of best fitness obtained so far, one value per generation (initial generation included). As in all the other evolution plots, each value is the median calculated over the 30 runs (see beginning of section).

The second figure is dedicated to size and contains two plots analogous to the previous. See, for example, Fig. 7. On the left (a), a boxplot of the mean tree size of run for each technique. On the right (b), the evolution of the mean tree size along the run. Tree size is the number of nodes of a tree. The mean tree size is the average number of nodes of the trees in the population. The mean tree size of run is the average of the mean tree size calculated over all the generations of the run. Comparative statements made on the results presented in the boxplot are also supported by statistical evidence.

The third and last figure contains four evolution plots. See, for example, Fig. 8. The first plot (a) shows the evolution of the percentage of inviable code in the individuals of the population. Inviable code is code that cannot contribute to the fitness of the individual, either because it is never executed or because its return value is always ignored. Non-inviable code is called effective code. Figure 5 shows the pseudo code of the (recursive) function that measures the amount of inviable code in a tree. The basic idea is to compare the fitness of the tree with the fitness of its branches, where a branch is a subtree rooted just below the root node of the tree. If a branch has the same fitness as the entire tree, then everything can be considered inviable code except (the effective code on) that branch. To maximize the amount of inviable code found in a tree, when several branches have the same fitness as the entire tree it is the one with the lowest amount of effective code that is chosen as the effective branch. To draw the plot, the percentage of inviable code in each individual is calculated, and then averaged for the entire population.

Still using Fig. 8, as example, the second plot (b) shows the evolution of the population diversity, measured as the percentage of structurally distinct individuals in the population (based on the variety measure [29]). The third plot (c) shows the evolution of the percentage of cloning due to limit restrictions. When using limits,

```
tree_size = number of nodes in tree

if tree_size = 1
   n_inviable = 0

else
   evaluate tree
   evaluate all the branches of tree

   if none of the branches has the same fitness as tree
      n_inviable = sum of the amount of inviable code in each branch

   else
      measure inviable code in branches with same fitness as tree
      my_branch = branch with lowest amount of effective code
      n_inviable = tree_size minus amount of effective code in my_branch
```

Fig. 5 Pseudo code of the recursive function that measures the number of inviable nodes (given by n_inviable) in a tree

either static or dynamic, whenever a new individual is rejected for breaking the limit, one of its parents is cloned and accepted into the new generation instead. The values in the plot represent the percentage of crossovers that resulted in a cloning operation.

The fourth and last plot (d) shows the evolution of the tree fill rate. The tree fill rate is calculated as the percentage of tree nodes relative to the number of nodes that would be expected in a random full tree, given its depth and the function and terminal sets used. Higher tree fill rates represent fuller trees. The average fill rate is calculated for the entire population. The expected number of nodes in a random full tree is given by

$$\sum_{i=0}^{d} avg\_kids^{i}$$

where d is the number of edges in the longest path between the root node and any terminal node, and *avg_kids* is calculated as

$$\frac{1}{|F|}\sum_{f\in F} arity_f$$

where F is the function set, and $arity_f$ is the number of arguments of function f.

5 Results

This section presents the results of all the experiments. Several plots and their brief descriptions are presented separately for each of the four problems studied.

5.1 Symbolic regression

Figure 6a shows a boxplot of the best fitness of run and the success rates achieved by each technique on the Regression problem, and (b) shows the evolution of the

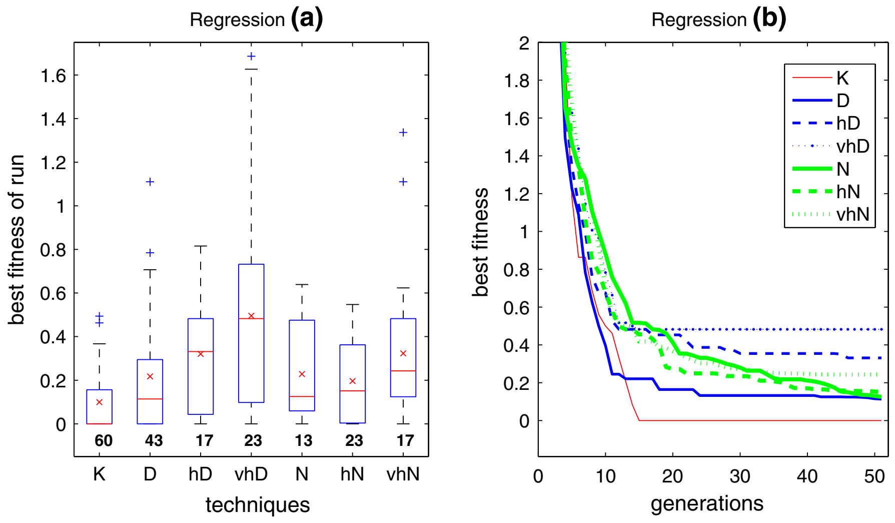

Fig. 6 Boxplot (**a**) and evolution curves (**b**) of the best fitness of run on the Regression problem. See Table 1 for the names of the techniques, and Table 4 for the *p*-values of the boxplot

best fitness along the run. The Koza (K) technique achieved the highest success rate, followed by DynDepth (D). Koza (K) also reached significantly better fitness of run than most of the other techniques, except DynDepth (D) and hDynNodes (hN). No other significant differences were observed.

Figure 7a shows a boxplot of the mean tree size of run for each technique on the Regression problem, and (b) shows the evolution of the mean tree size along the run.

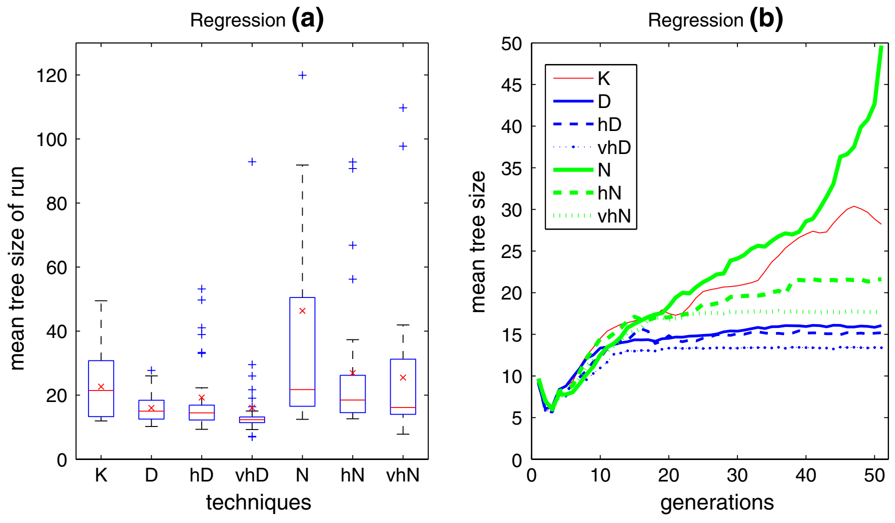

Fig. 7 Boxplot (**a**) and evolution curves (**b**) of the mean tree size on the Regression problem. See Table 1 for the names of the techniques, and Table 4 for the *p*-values of the boxplot

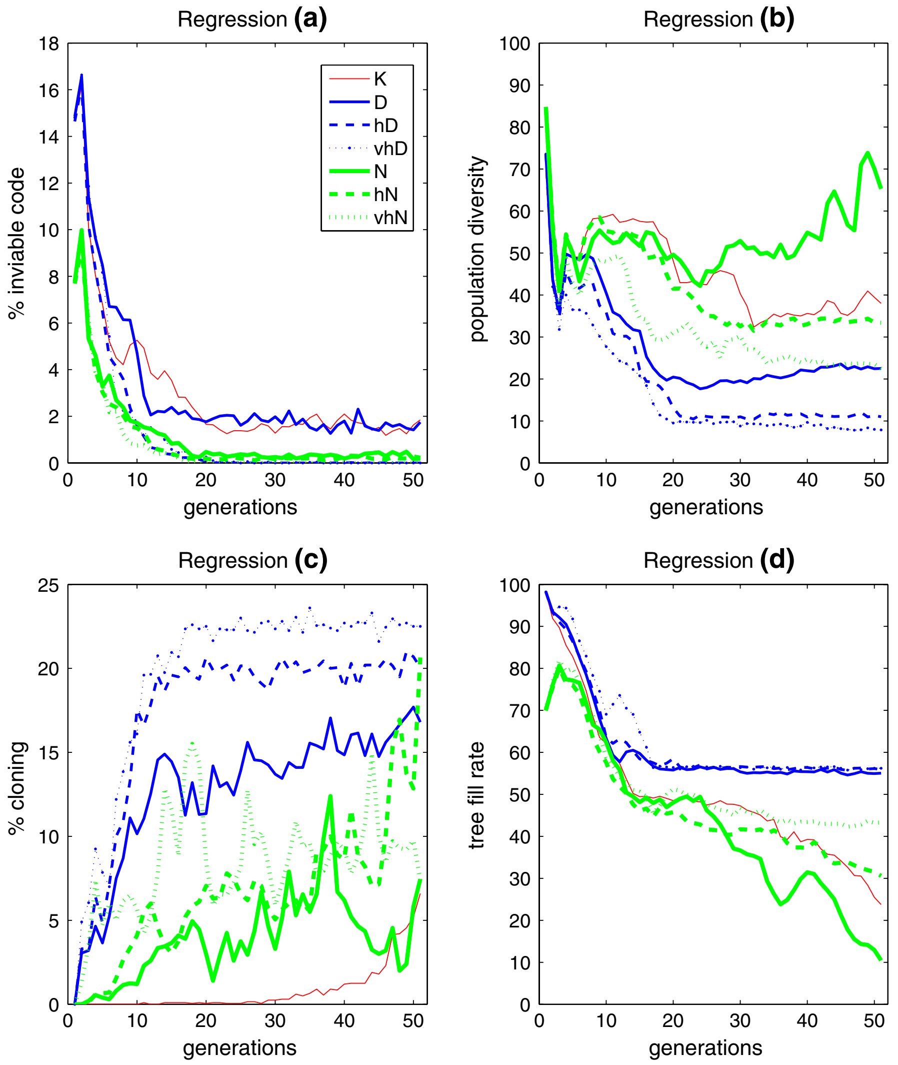

Fig. 8 Percentage of inviable code, population diversity, percentage of cloning, and tree fill rate on the Regression problem. See Table 1 for the names of the techniques

Both DynDepth (D) and vhDynDepth (vhD) used significantly smaller trees than Koza (K) and most of the size variants. No other significant differences were observed.

Figure 8a shows the evolution of the percentage of inviable code on the Regression problem. The results support the notion that this problem is not prone to the propagation of inviable code (Sect. 4.1.1). The small percentage of inviable code present in the initial random individuals quickly dropped to values close to zero. Only Koza (K) and DynDepth (D) maintained a slightly higher level of inviable code, curiously the two techniques that ranked first in best fitness and presented the highest success rates of all.

The second plot of Fig. 8b shows the evolution of the population diversity. Beginning with 75 to 85% of structurally distinct individuals in the population, this diversity quickly dropped in the beginning of the run, and after a slight recuperation it continued dropping until it roughly stabilized in different values for each technique, with the depth variants presenting the lowest values, around 10 to 20%. Only DynNodes (N) was able to increase the population diversity beyond the values reached in the initial drop, ending the run with roughly 70%.

The third plot (c) shows the evolution of the percentage of cloning caused by limit restrictions. The results show that the depth variants performed more cloning operations than the other techniques, which may account for the lower population diversity observed on the previous plot. All the dynamic techniques performed cloning operations throughout the entire run. As expected, the Koza (K) technique did not perform cloning until later in the run, when its static limit was finally reached.

The last plot (d) shows the evolution of the tree fill rate. The size variants adopted sparser trees than the depth variants, a phenomenon that was already expected considering the nature of the limits used. Unlike depth limits, size limits do not constrain the shape of the trees. The Koza (K) technique did not suffer from much shape constraints either, as its static limit was only reached relatively late in the run (see previous plot).

5.2 Artificial ant

Figure 9a shows a boxplot of the best fitness of run and the success rates achieved by each technique on the Artificial Ant problem, and (b) shows the evolution of the

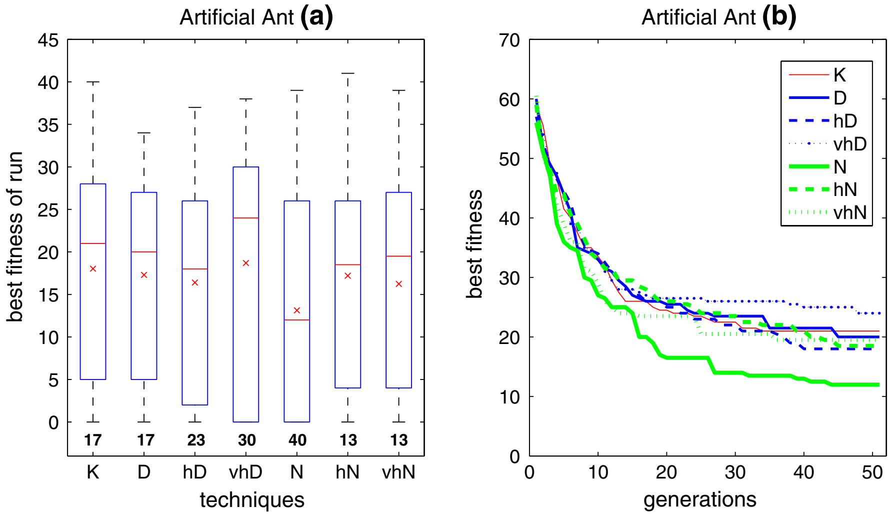

Fig. 9 Boxplot (**a**) and evolution curves (**b**) of the best fitness of run on the Artificial Ant problem. See Table 1 for the names of the techniques, and Table 5 for the p-values of the boxplot

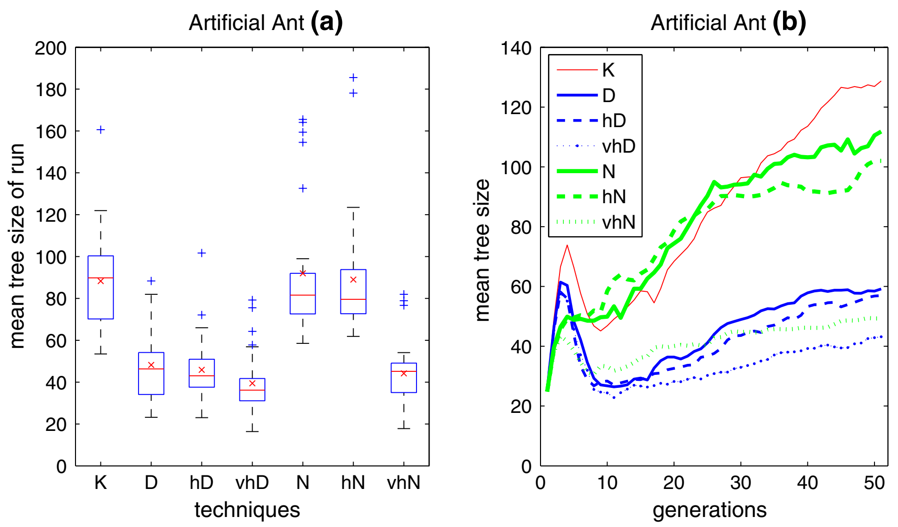

Fig. 10 Boxplot (**a**) and evolution curves (**b**) of the mean tree size on the Artificial Ant problem. See Table 1 for the names of the techniques, and Table 5 for the *p*-values of the boxplot

best fitness along the run. The highest success rate was achieved by DynNodes (N), the technique that also reached the best fitness along the run, although not significantly. In fact, there were no significant differences in best fitness of run between any of the studied techniques.

Figure 10a shows a boxplot of the mean tree size of run for each technique on the Artificial Ant problem, and (b) shows the evolution of the mean tree size along the run. All the dynamic depth variants, as well as vhDynNodes (vhN), used significantly smaller trees than the other techniques. There were no significant differences between Koza (K), DynNodes (N), and hDynNodes (hN), or between the depth variants.

Figure 11a shows the evolution of the percentage of inviable code on the Artificial Ant problem. Knowing that this problem is very prone to inviable code (Sect. 4.1.2), it is no surprise that, from an initial percentage of around 80%, the amount of inviable code hardly dropped below 60%, reaching as high as 85% by the end of the run. DynNodes (N) was the technique that reached higher percentages of inviable code, followed closely by hDynNodes (hN). It is interesting to verify that, as in the Regression problem, the highest level of inviable code belongs to the technique achieving the highest success rate. The depth variants were the least prone to inviable code, maybe the reason why they managed to use significantly smaller trees.

The second plot (b) shows the evolution of the population diversity, where two distinct behaviors can be observed. Koza (K) and two of the size variants, DynNodes (N) and hDynNodes (hN), quickly increased the population diversity from its initial value of around 75%, reaching as high as 95% by the end of the run. The depth variants, accompanied by vhDynNodes (vhN), dropped the population diversity in the first half of the run, and then steadily increased it until reaching 80 to 90%.

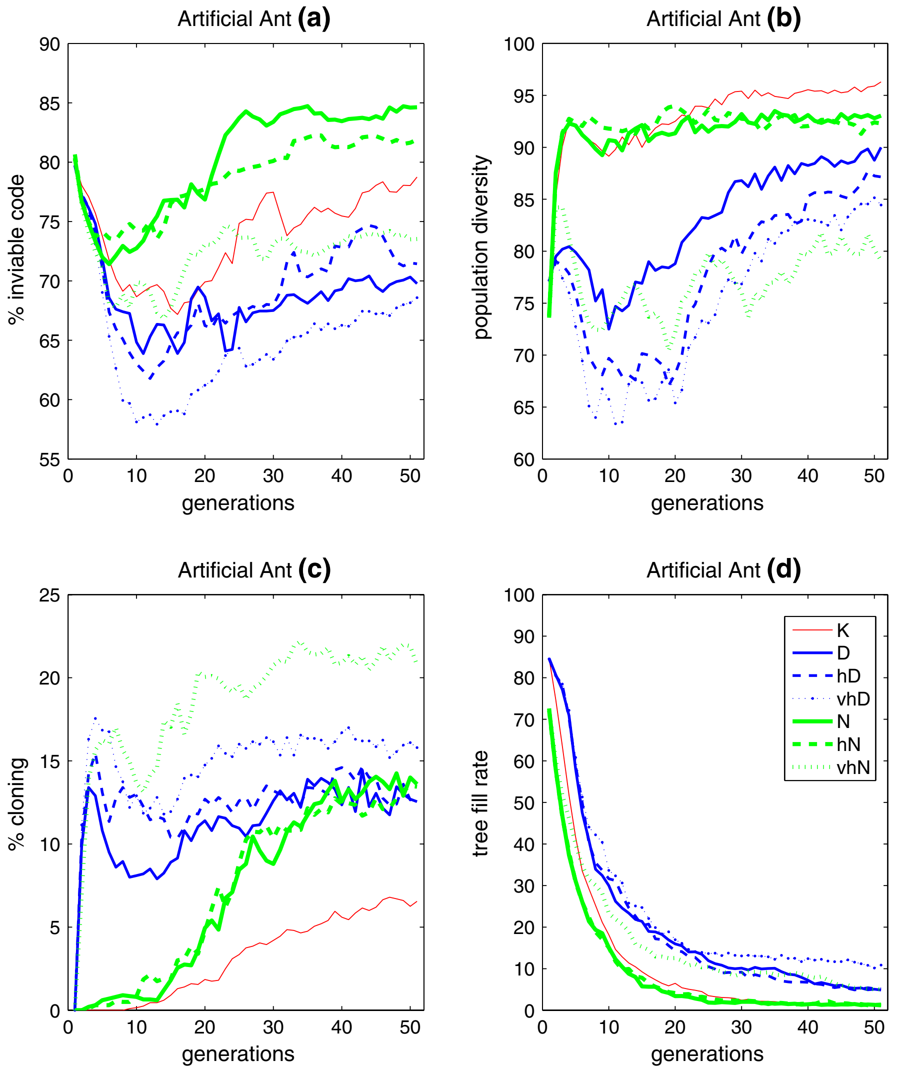

Fig. 11 Percentage of inviable code, population diversity, percentage of cloning, and tree fill rate on the Artificial Ant problem. See Table 1 for the names of the techniques

The third plot (c) shows the evolution of the percentage of cloning caused by limit restrictions, where once more different behaviors can be observed. The depth variants, once again accompanied by vhDynNodes (vhN), performed intensive cloning right from the beginning of the run. Koza (K) and the remaining size techniques performed very little cloning in the beginning of the run, and then steadily increased its frequency until the end, but the curve of the size techniques is much more steep. As in the Regression problem, a higher amount of cloning operations seems to be associated with a lower population diversity.

The last plot (d) shows the evolution of the tree fill rate. All the techniques quickly dropped the fill rate right from the start, reaching very low values by the end of the run.

5.3 5-Bit even parity

Figure 12a shows a boxplot of the best fitness of run and the success rates achieved by each technique on the Parity problem, and (b) shows the evolution of the best fitness along the run. The success rates were all null, as none of the techniques was ever able to find an optimal solution. In terms of best fitness of run, all the size variants did worse than Koza (K). No other significant differences were observed.

Figure 13a shows a boxplot of the mean tree size of run for each technique on the Parity problem, and (b) shows the evolution of the mean tree size along the run. All the dynamic techniques used significantly smaller trees than Koza (K). Among the dynamic depth techniques, the very heavy variant used significantly smaller trees than DynDepth (D). Both hDynNodes (hN) and vhDynNodes (vhN) also used significantly smaller trees than DynDepth (D). No other significant differences were observed.

Figure 14a shows the evolution of the percentage of inviable code on the Parity problem. There was relatively little inviable code in this problem, with initial values of 25 to 30% and many variations along the run, but never reaching higher than 30%.

The second plot (b) shows the evolution of the population diversity. All the techniques were able to increase their initial diversity and maintain very high values throughout the run (between 85 and 100%). Koza (K) was the technique sustaining

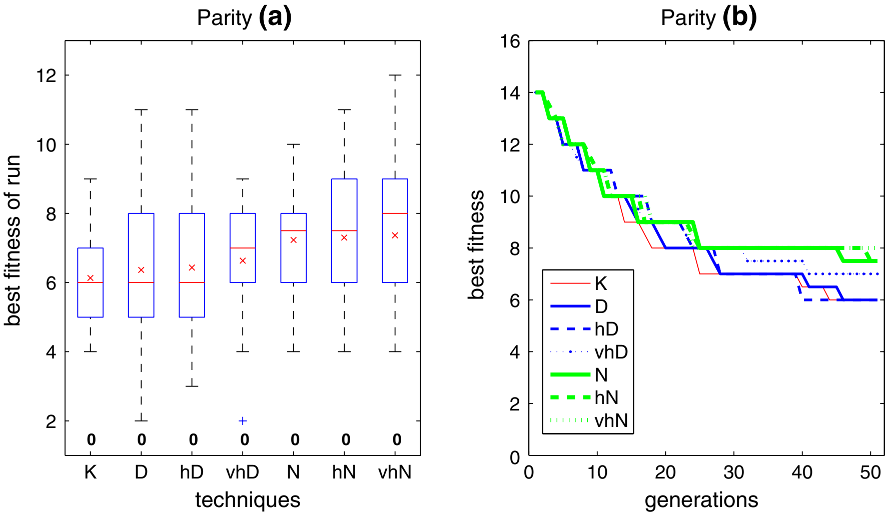

Fig. 12 Boxplot (**a**) and evolution curves (**b**) of the best fitness of run on the Parity problem. See Table 1 for the names of the techniques, and Table 6 for the p-values of the boxplot

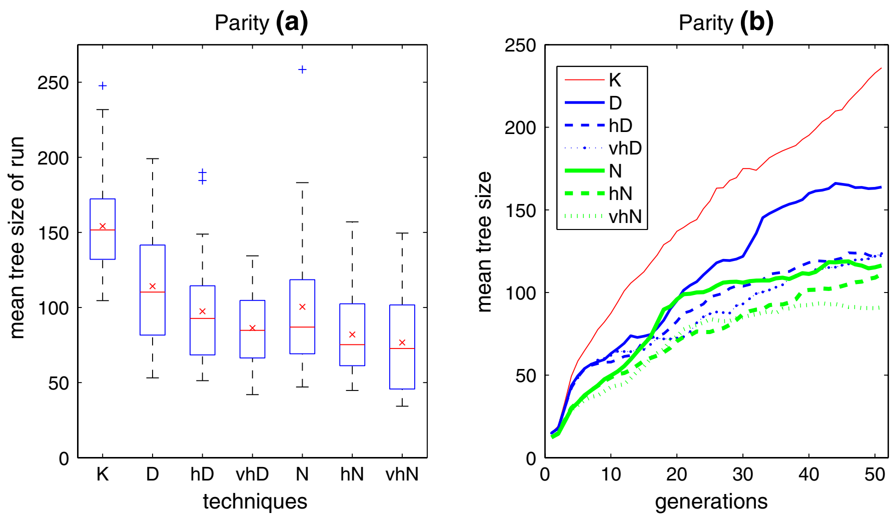

Fig. 13 Boxplot (**a**) and evolution curves (**b**) of the mean tree size on the Parity problem. See Table 1 for the names of the techniques, and Table 6 for the *p*-values of the boxplot

the highest diversity values, followed by the depth variants, and finally the size variants.

The third plot (c) shows the evolution of the percentage of cloning caused by limit restrictions. During most of the run it was the size variants that performed the highest number of cloning operations, followed by the depth variants, and finally the Koza (K) technique. As in the previous two problems, more cloning seems to be related to lower population diversity.

The last plot (d) shows the evolution of the tree fill rate. As in the Artificial Ant problem, all the techniques quickly drop the fill rate right from the beginning of the run, reaching extremely low values by the end of the run.

5.4 11-Bit Boolean multiplexer

Figure 15a shows a boxplot of the best fitness of run and the success rates achieved by each technique on the Multiplexer problem, and (b) shows the evolution of the best fitness along the run. The highest success rate was achieved by the Koza (K) technique, followed by vhDynDepth (vhD). In terms of best fitness of run, all the size variants did worse than Koza (K), and vhDynNodes (vhN) also did worse than the depth variants. No other significant differences were observed.

Figure 16a shows a boxplot of the mean tree size of run for each technique on the Multiplexer problem, and (b) shows the evolution of the mean tree size along the run. All the dynamic techniques used significantly smaller trees than Koza (K), except DynNodes (N) that showed no significant difference. While there are no significant differences among the depth variants, all the size variants were significantly different from each other. Both hDynDepth (hD) and vhDynDepth (vhD) were identical to their size counterparts hDynNodes (hN) and vhDynNodes (vhN).

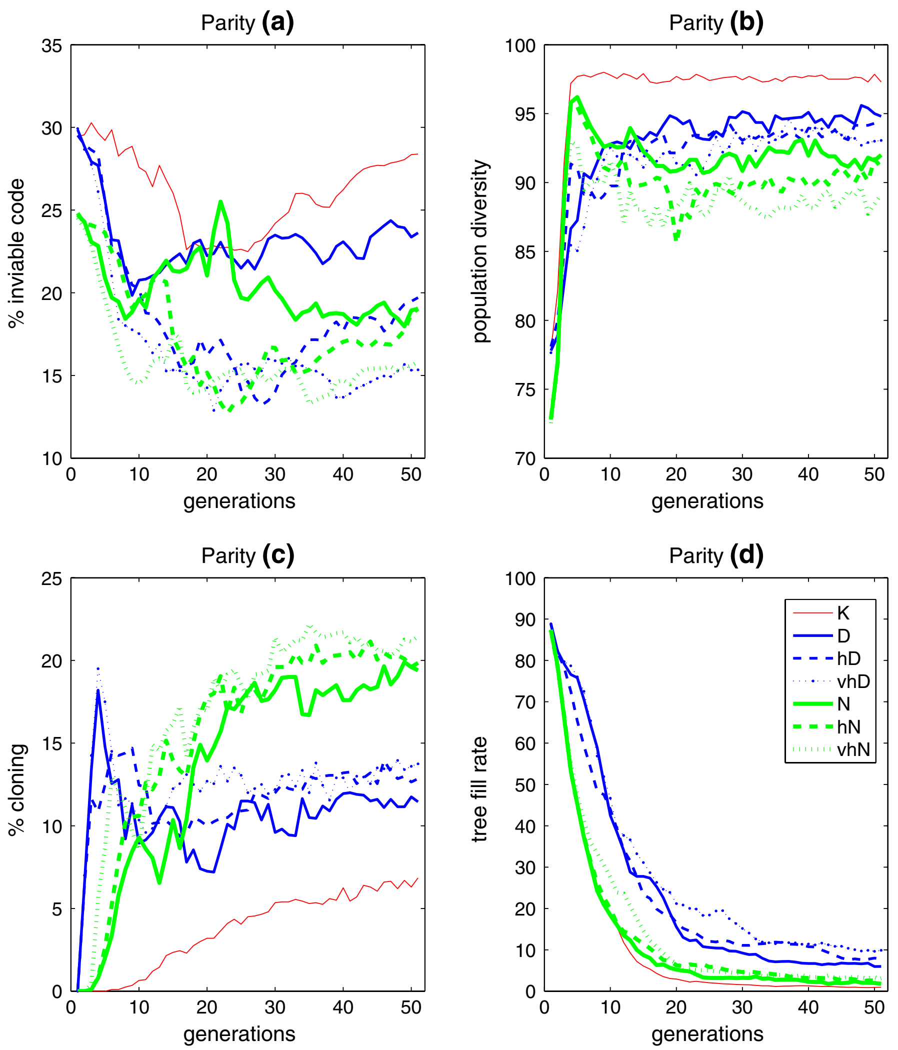

Fig. 14 Percentage of inviable code, population diversity, percentage of cloning, and tree fill rate on the Parity problem. See Table 1 for the names of the techniques

Figure 17a shows the evolution of the percentage of inviable code on the Multiplexer problem. As expected (Sect. 4.1.4), the percentage of inviable code in this problem was relatively low. From the initial percentages of 10 to 12%, only DynNodes (N) substantially increased it, with the remaining techniques finishing the run with only 12 to 14% of inviable code. Unlike what happened with the Regression and Artificial Ant problems, the technique with the largest amount of inviable code was not among the most successful.

The second plot (b) shows the evolution of the population diversity. After an initial drop, all the techniques increased their diversity and maintained extremely high

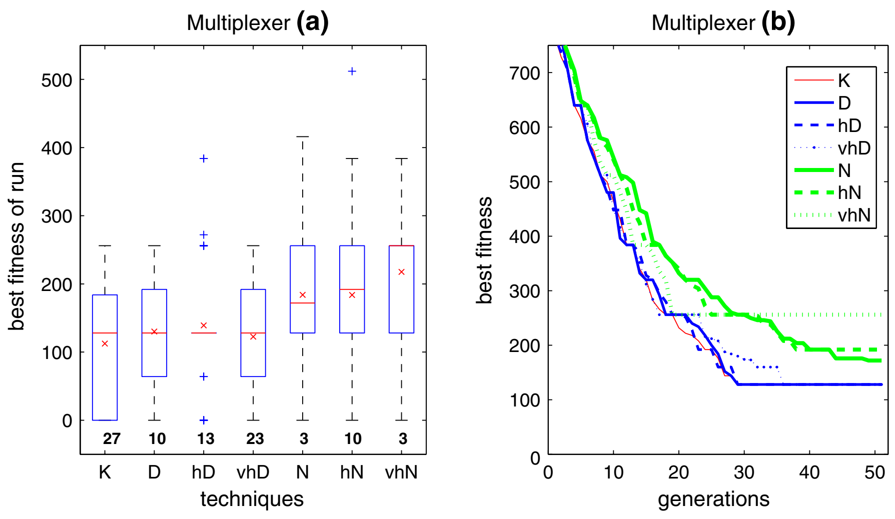

Fig. 15 Boxplot (**a**) and evolution curves (**b**) of the best fitness of run on the Multiplexer problem. See Table 1 for the names of the techniques, and Table 7 for the *p*-values of the boxplot

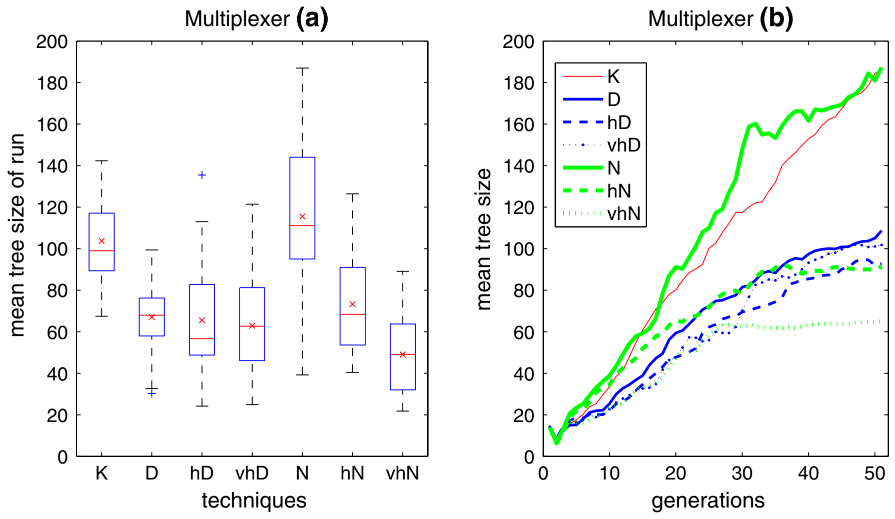

Fig. 16 Boxplot (**a**) and evolution curves (**b**) of the mean tree size on the Multiplexer problem. See Table 1 for the names of the techniques, and Table 7 for the *p*-values of the boxplot

values throughout the run (between 90 and 100%), as in the Parity problem. Koza (K) was the technique sustaining the highest diversity values, followed by the depth variants accompanied by DynNodes (N), and finally the remaining size variants.

The third plot (c) shows the evolution of the percentage of cloning caused by limit restrictions. Once more an apparent relationship between cloning and diversity stands out, with Koza (K) performing the least number of cloning operations, followed by the depth variants and DynNodes (N), and finally the remaining size variants.

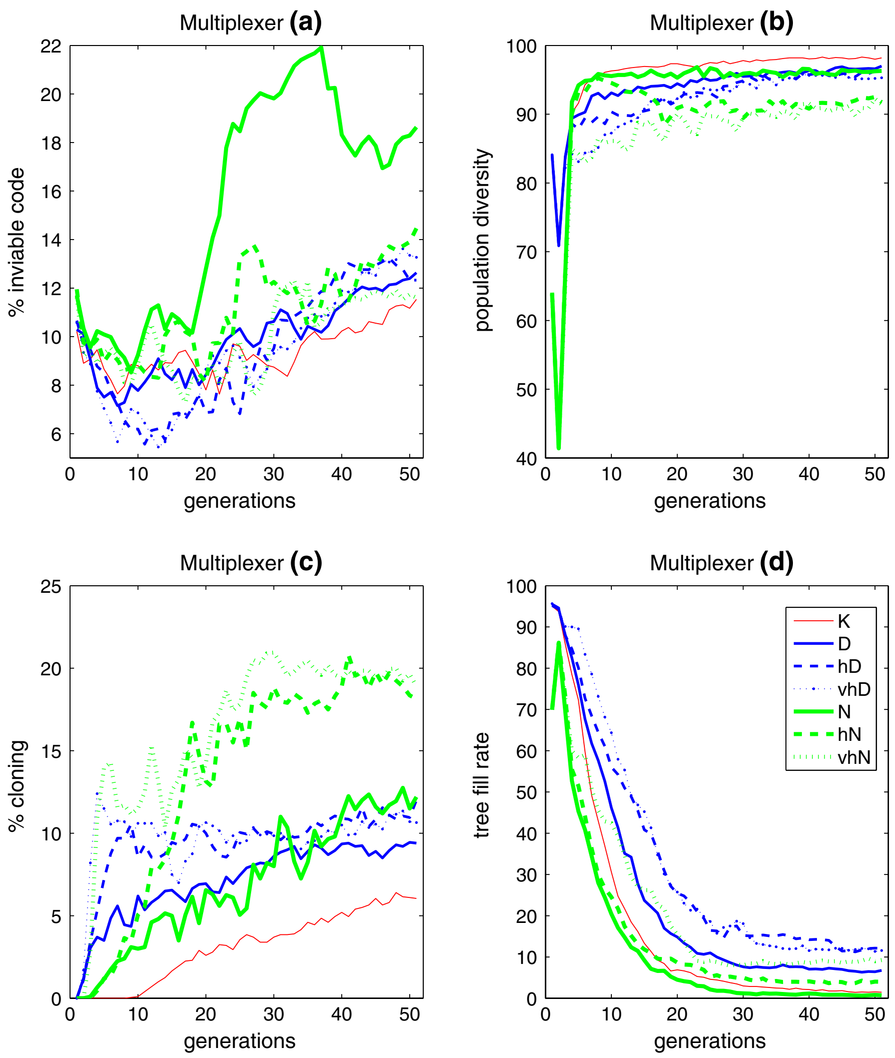

Fig. 17 Percentage of inviable code, population diversity, percentage of cloning, and tree fill rate on the Multiplexer problem. See Table 1 for the names of the techniques

The last plot (d) shows the evolution of the tree fill rate. As in the previous two problems, all the techniques quickly dropped the fill rate (after an initial increase by the size variants) and reached extremely low values by the end of the run.

6 Discussion

This section summarizes and discusses the results obtained. It also debates the general usage and implementation details of depth and size limits in GP.

6.1 Summary and quality of the results

The quality of the results obtained with the Dynamic Limits approach revealed to be highly dependent on the type of limit used: depth or size. The depth limits performed very well across the set of problems studied. In all except the Regression problem, all the depth variants were able to achieve similar fitness to Koza using significantly smaller trees. In the Regression problem, one of the variants (DynDepth) was also able to attain the same achievement. Only in one of the problems (Artificial Ant) does this tree size reduction seem to be related to a lower percentage of inviable code. Curiously enough, in two of the problems (Regression and Artificial Ant) the highest percentages of inviable code belonged to the most successful techniques. There seems to be no real advantage in using the heavy variants. In one of the problems (Parity) the heavy and very heavy depth provided significantly lower mean tree size without impairing fitness, but on another (Regression) they caused the best fitness of run to be significantly worse than the non-heavy variant.

The dynamic limits on size did not perform so well. Most of the size variants achieved significantly worse fitness than Koza or the depth variants in the majority of the problems. There was only a modest success on the Artificial Ant problem, where one of the variants (DynNodes) had a higher success rate than the rest, but with no significant improvement in the best fitness achieved. Furthermore, on the few cases when the size variants reached similar fitness to Koza and the depth variants, they failed to attain significantly lower mean tree size. Although it is not possible to directly compare the present size limits with their previous implementation [66], due to the usage of different parameters, it is clear that the new implementation did not improve the results.

The amount of cloning operations performed due to depth or size restrictions seems to be inversely related to the population diversity, which is not surprising. However, the results showed no evidence that the techniques maintaining higher diversity achieve better fitness than the ones with lower diversity. In terms of tree fill rate, the depth variants always obtained higher values, which was also expected from the nature of their limits. The size variants always behaved very similarly to the Koza technique. The tree fill rate does not seem to influence the performance of the techniques, since different behaviors in the evolution of tree fill rate (particularly in the Regression problem) yielded similar results in terms of best fitness of run.

There are obvious differences in the initial values of cloning, diversity, and fill rate of the size variants, caused by the specific population initialization adopted for size limits (Sect. 3.2.2). Although tempting it may be to blame the poor performance of the size limits on the fact that they are subject to different initial conditions than the other techniques, the results have not suggested that either of the indicators (inviable code, diversity, fill rate) are determinant in the success of the different techniques. Furthermore, the observation of the plots suggests that, during the first few generations, most of the initial differences are quickly “corrected” and overridden by the particular behavior produced by the size restrictions.

All in all, the dynamic depth limits were very successful at controlling bloat. In particular, the DynDepth technique never failed to achieve the same fitness as Koza, while using significantly smaller trees. Furthermore, this was accomplished without relying on any static limit. On the other hand, none of the studied indicators (amount of inviable code, population diversity, amount of cloning, tree fill rate) seem to provide an explanation to why the size limits, both old and new implementations, failed in most of the problems.

6.2 Considerations on the usage of limit restrictions

There is a considerable amount of concern regarding the usage of depth or size limits in GP. Studies have suggested that the usage of size or depth limits can interfere with search efficiency once the average program size approaches the limit, leading to premature convergence [36, Chap. 10; 20, 34]. Another study deals with the impact of size limits on the average size of the individuals in the population [47], and recent work related to the crossover bias theory reports that size limits actually speed code growth in the early stages of the run [16] (see Sect. 2.6). The most traditional way of implementing the limits is to reject the invalid individuals and replace them with one of their parents. Although this effectively prevents the individuals from growing too large, the replication of the parents may have undesirable effects. The larger parents are the ones that usually produce invalid offspring, so they tend to be replicated more often than the smaller parents. The population is filled with the largest individuals, and quickly rushes to the limit.

Alternative ways of implementing the depth or size limits are: (1) retry the genetic operator until a valid offspring is produced, either with the same parents or using different ones; (2) accept the invalid individuals but give them such bad fitness values that they will not be selected for reproduction in the next generation. In the light of the crossover bias theory (Sect. 2.6), retrying the genetic operator may not be advisable, since it provides another opportunity for the creation of more small unfit individuals. Accepting the large invalid offspring seems like a better measure against the undesirable crossover bias, since the large nullified individuals will never reproduce. However, in the presence of a dynamic and highly constrained limit that does not rise easily, parent replication may still provide advantages over these other options.

In the present implementation of Dynamic Limits, the traditional parent replication is indeed the action taken when the offspring violate the limit. But, unlike typical (static) limits, the initial dynamic limit is very low, as low as the maximum depth/size of the initial trees, and it will not be increased until a deeper/longer individual proves to be better than any other found so far. This highly constrains the search space, and for most problems it is known that the good solutions lie somewhere beyond this limit, not below. When a larger and better individual pushes the limit up, it means the process has entered a better searching ground—better solutions can be found within the new allowed depth/size. So, rushing the population towards this better, but still highly constrained, search space, may actually speed the convergence to better solutions. Parent replication along with a slowly increasing limit does not necessarily entail the drawbacks of using a high static limit. This is supported by the quality of the results obtained and discussed previously.

7 Conclusions

We have reviewed the past and current theories regarding the reasons why bloat occurs in GP, and introduced the concept of Dynamic Limits, a new approach to bloat control. Dynamic Limits is inspired by the most traditional technique of imposing a fixed limit on the depth of the individuals allowed in the population, introduced by Koza in tree-based GP. It implements a dynamic limit that can be raised or lowered, depending on the best solution found so far, and can be applied either to the depth or size of the programs being evolved.

Four different problems were used as a benchmark to study the efficiency of Dynamic Limits. The quality of the results obtained revealed to be highly dependent on the type of limit used: depth or size. The depth limits performed very well across the set of problems studied, with one of its variants always achieving similar fitness to the Koza technique, while using significantly smaller trees. The size limits did not perform so well, most of the times obtaining significantly worse fitness than Koza or the depth limit variants.

A strong bloat control method should be able to deal with any type of problem, and be quite insensitive to the choice of parameters and even the combination of algorithmic elements like the evaluation, selection, and breeding procedures. The Dynamic Limits follow these criteria, and the dynamic depth techniques have proven to be very successful at controlling bloat. Unlike many other methods available so far, this new approach does not require specific genetic operators, modifications in fitness evaluation or different selection schemes, nor does it add any parameters to the search process. Furthermore, its implementation is very simple and can be coupled with other bloat control techniques. Finally, it does *not* rely on the usage of a static upper limit to obtain high quality results, which is a very desirable property.

8 Future work

As future work it would be interesting to adopt alternative actions when an individual violates the limit. Instead of replicating one of the parents into the next generation, the options discussed in Sect. 6.2 could be tested: (1) retrying the genetic operator until a valid offspring is produced, either with the same parents or using different ones; (2) accepting the invalid individuals but null their fitness so they will not be selected for reproduction in the next generation. In the context of the Dynamic Limits approach, the results of such experiments could reveal the dynamics and expose the factors that are allowing the dynamic depth to produce such good results, and eventually provide some insight on how to improve the dynamic size. In the context of the newest bloat theory (Sect. 2.6), the experiments could be used to validate and study the role played by the crossover bias on the emergence of bloat, by measuring how much it varies in the presence of alternative offspring survival schemes.

Acknowledgements This work was partially supported by grant QLK2-CT-2000-01020 from the European Commission and grants SFRH/BD/14167/2003 and POCTI/1999/BSE/34794 from Fundação para a Ciência e a Tecnologia, Portugal. We would like to thank Jonas Almeida (University of Texas MD Anderson Cancer Center, USA) for his collaboration in the early stages of this research, and Riccardo Poli (University of Essex, UK) for fruitful discussions leading to a substantial improvement of the manuscript.

Appendix

Table 4 *p*-Values concerning the mean tree size (top right half) and best fitness of run (bottom left half) on the Regression problem, using pairwise non-parametric ANOVAs

	Mean tree size							
	K	D	hD	vhD	N	hN	vhN	
K		**0.0093**	0.0493	**0.0000**	0.1008	0.7117	0.6574	K
D	0.1168		0.8941	0.0141	**0.0001**	**0.0039**	0.0736	D
hD	**0.0003**	0.0530		0.0345	**0.0004**	**0.0089**	0.1103	hD
vhD	**0.0001**	0.0117	0.2537		**0.0000**	**0.0000**	**0.0003**	vhD
N	**0.0030**	0.3020	0.3541	0.0446		0.1558	0.0333	N
hN	0.0106	0.4694	0.1372	0.0267	0.6402		0.4688	hN
vhN	**0.0003**	0.0685	0.9232	0.1876	0.2827	0.1518		vhN
	K	D	hD	vhD	N	hN	vhN	
Best fitness								

Statistical significance is considered where $p < 0.01$ (in bold). See Figs. 6–7a for the boxplots. See Table 1 for the names of the techniques

Table 5 *p*-Values concerning the mean tree size (top right half) and best fitness of run (bottom left half) on the Artificial Ant problem, using pairwise non-parametric ANOVAs

	Mean tree size							
	K	D	hD	vhD	N	hN	vhN	
K		**0.0000**	**0.0000**	**0.0000**	0.9058	0.8941	**0.0000**	K
D	0.8997		0.4965	0.0256	**0.0000**	**0.0000**	0.3912	D
hD	0.5725	0.6939		0.0195	**0.0000**	**0.0000**	0.9176	hD
vhD	0.8060	0.6659	0.4506		**0.0000**	**0.0000**	0.1316	vhD
N	0.1127	0.1641	0.3398	0.1446		0.6898	**0.0000**	N
hN	0.7726	0.8822	0.7106	0.6340	0.1765		**0.0000**	hN
vhN	0.6352	0.7054	0.8528	0.5568	0.2204	0.9527		vhN
	K	D	hD	vhD	N	hN	vhN	
Best fitness								

Statistical significance is considered where $p < 0.01$ (in bold). See Figs. 9–10a for the boxplots. See Table 1 for the names of the techniques

Table 6 *p*-Values concerning the mean tree size (top right half) and best fitness of run (bottom left half) on the Parity problem, using pairwise non-parametric ANOVAs

	Mean tree size							
	K	D	hD	vhD	N	hN	vhN	
K		**0.0001**	**0.0000**	**0.0000**	**0.0000**	**0.0000**	**0.0000**	K
D	0.7452		0.0891	**0.0081**	0.1071	**0.0010**	**0.0004**	D
hD	0.9879	0.8926		0.3516	0.8245	0.0690	0.0219	hD
vhD	0.1172	0.4805	0.3706		0.3831	0.3077	0.1137	vhD
N	**0.0049**	0.0787	0.0637	0.1973		0.1316	0.0298	N
hN	**0.0075**	0.0773	0.0590	0.1954	0.9222		0.4508	hN
vhN	**0.0093**	0.0792	0.0719	0.1829	0.8391	0.9343		vhN
	K	D	hD	vhD	N	hN	vhN	
Best fitness								

Statistical significance is considered where $p < 0.01$ (in bold). See Figs. 12–13a for the boxplots. See Table 1 for the names of the techniques

Table 7 *p*-Values concerning the mean tree size (top right half) and best fitness of run (bottom left half) on the Multiplexer problem, using pairwise non-parametric ANOVAs

	Mean tree size							
	K	D	hD	vhD	N	hN	vhN	
K		**0.0000**	**0.0000**	**0.0000**	0.0863	**0.0000**	**0.0000**	K
D	0.4589		0.2871	0.3516	**0.0000**	0.6361	**0.0006**	D
hD	0.4420	0.8755		0.9176	**0.0000**	0.1882	0.0141	hD
vhD	0.7118	0.7592	0.6246		**0.0000**	0.1646	0.0228	vhD
N	**0.0068**	0.0294	0.0445	0.0171		**0.0000**	**0.0000**	N
hN	**0.0093**	0.0356	0.0542	0.0289	0.9045		**0.0002**	hN
vhN	**0.0001**	**0.0005**	**0.0027**	**0.0004**	0.2412	0.1125		vhN
	K	D	hD	vhD	N	hN	vhN	
Best fitness								

Statistical significance is considered where $p < 0.01$ (in bold). See Figs. 15–16a for the boxplots. See Table 1 for the names of the techniques

References

1. L. Altenberg, The evolution of evolvability in genetic programming, in *Advances in Genetic Programming*, ed. by K.E. Kinnear Jr. (MIT Press, Cambridge, MA, 1994), pp. 47–74
2. L. Altenberg, Emergent phenomena in genetic programming, in *Proceedings of the 3rd Conference on Evolutionary Programming*, ed. by A.V. Sebald, L.J. Fogel (World Scientific Publishing, River Edge, NJ, 1994), pp. 233–241
3. D. Andre, A. Teller, A study in program response and the negative effects of introns in genetic programming, in *Proceedings of GP'96*, ed. by J.R. Koza et al. (MIT Press, Cambridge, MA, 1996), pp. 28–31

4. P.J. Angeline, Genetic programming and emergent intelligence, in *Advances in Genetic Programming*, ed. by K.E. Kinnear Jr. (MIT Press, Cambridge, MA, 1994), pp. 75–98
5. P.J. Angeline, Two self-adaptive crossover operators for genetic programming, in *Advances in Genetic Programming 2*, ed. by P.J. Angeline, K.E. Kinnear Jr. (MIT Press, Cambridge, MA, 1996), pp. 89–110
6. P.J. Angeline, A historical perspective on the evolution of executable structures. Fundam. Informaticae **35**(1–4), 179–195 (1998)
7. W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone, *Genetic Programming—An Introduction* (dpunkt.verlag and Morgan Kaufmann, Heidelberg and San Francisco, CA, 1998)
8. W. Banzhaf, W.B. Langdon, Some considerations on the reason for bloat. Genet. Program. Evolvable Mach. **3**(1), 81–91 (2002)
9. W. Banzhaf, F.D. Francone, P. Nordin, Some emergent properties of variable size EAs. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97 (1997)
10. T. Blickle, Theory of evolutionary algorithms and applications to system design. PhD thesis, Swiss Federal Institute of Technology, Computer Engineering and Networks Laboratory (1996)
11. T. Blickle, L. Thiele, Genetic programming and redundancy, in *Genetic Algorithms within the Framework of Evolutionary Computation*, ed. by J. Hopf (Max-Planck-Institut für Informatik, Saarbriicken, 1994), pp. 33–38
12. M. Brameier, W. Banzhaf, Neutral variations cause bloat in linear GP, in *Proceedings of EuroGP-2003*, ed. by C. Ryan et al. (Springer, Berlin, 2003), pp. 286–296
13. J. Cuendet, Populations dynamiques en programmation génétique. MSc thesis, Université de Lausanne, Université de Genève (2004)
14. L.E. Da Costa, J.A. Landry, Relaxed genetic programming, in *Proceedings of GECCO-2006*, ed. by M. Keijzer et al. (ACM Press, New York, NY, 2006), pp. 937–938
15. S. Dignum, R. Poli, Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat, in *Proceedings of GECCO-2007*, ed. by D. Thierens et al. (ACM Press, New York, NY, 2007), pp. 1588–1595
16. S. Dignum, R. Poli, Crossover, sampling, bloat and the harmful effects of size limits, in *Proceedings of EuroGP-2008*, ed. by M. O'Neill et al. (Springer, Berlin, 2008), pp. 158–169
17. A. Ekart, S.Z. Németh, Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. Genet. Program. Evolvable Mach. **2**(1), 61–73 (2001)
18. F. Fernandez, L. Vanneschi, M. Tomassini, The effect of plagues in genetic programming: a study of variable-size populations, in *Proceedings of EuroGP-2003*, ed. by C. Ryan et al. (Springer, Berlin, 2003), pp. 317–326
19. F. Fernandez, M. Tomassini, L. Vanneschi, Saving computational effort in genetic programming by means of plagues, in *Proceedings of CEC-2003*, ed. by R. Sarker et al. (IEEE Press, Piscataway, NJ, 2003), pp. 2042–2049
20. C. Gathercole, P. Ross, An adverse interaction between crossover and restricted tree depth in genetic programming, in *Proceedings of GP'96*, ed. by J.R. Koza et al. (MIT Press, Cambridge, MA, 1996), pp. 291–296
21. S. Gelly, O. Teytaud, N. Bredeche, M. Schoenauer, A statistical learning theory approach of bloat, in *Proceedings of GECCO-2005*, ed. by H.-G. Beyer et al. (ACM Press, New York, NY, 2005), pp. 1783–1784
22. S. Gelly, O. Teytaud, N. Bredeche, M. Schoenauer, Universal consistency and bloat in GP. Rev. Intell. Artif. **20**(6), 805–827 (2006)
23. S. Gustafson, A. Ekart, E. Burke, G. Kendall, Problem difficulty and code growth in genetic programming. Genet. Program. Evolvable Mach. **5**(3), 271–290 (2004)
24. C. Igel, K. Chellapilla, Investigating the influence of depth and degree of genotypic change on fitness in genetic programming, in *Proceedings of GECCO-1999*, ed. by W. Banzhaf et al. (Morgan Kaufmann, San Francisco, CA, 1999), pp. 1061–1068
25. K. Janardan, Weighted Lagrange distributions and their characterizations. SIAM J. Appl. Math. **47**(2), 411–415 (1987)
26. K. Janardan, B. Rao, Lagrange distributions of the second kind and weighted distributions. SIAM J. Appl. Math. **43**(2), 302–313 (1983)
27. K.E. Kinnear Jr., Generality and difficulty in genetic programming: evolving a sort, in *Proceedings of ICGA'93*, ed. by S. Forrest (Morgan Kaufmann, San Francisco, CA, 1993), pp. 287–294

28. J.R. Koza, *Genetic Programming – On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, MA, 1992)
29. W.B. Langdon, *Genetic Programming + Data Structures = Automatic Programming!* (Kluwer Academic Publishers, Boston, MA, 1998)
30. W.B. Langdon, The evolution of size in variable length representations, in *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation* (IEEE Press, Piscataway, NJ, 1998), pp. 633–638
31. W.B. Langdon, Size fair and homologous tree genetic programming crossovers. Genet. Program. Evolvable Mach. **1**(1/2), 95–119 (2000)
32. W.B. Langdon, Quadratic bloat in genetic programming, in *Proceedings of GECCO-2000*, ed. by D. Whitley et al. (Morgan Kaufmann, San Francisco, CA, 2000), pp. 451–458
33. W.B. Langdon, R. Poli, Fitness causes bloat, in *Proceedings of the Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, ed. by P.K. Chawdhry et al. (Springer, Berlin, 1997), pp. 13–22
34. W.B. Langdon, R. Poli, An analysis of the MAX problem in genetic programming, in *Proceedings of GP'97*, ed. by J.R. Koza et al. (Morgan Kaufman, San Francisco, CA, 1997), pp. 222–230
35. W.B. Langdon, R. Poli, Fitness causes bloat: mutation, in *Proceedings of EuroGP'98*, ed. by W. Banzhaf et al. (Springer, Berlin, 1998), pp. 37–48
36. W.B. Langdon, R. Poli, *Foundations of Genetic Programming* (Springer, Berlin, 2002)
37. W.B. Langdon, T. Soule, R. Poli, J.A. Foster, The evolution of size and shape, in *Advances in Genetic Programming 3*, ed. by L. Spector et al. (MIT Press, Cambridge, MA, 1999), pp. 163–190
38. W.B. Langdon, W. Banzhaf, Genetic programming bloat without semantics, in *Proceedings of PPSN-2000*, ed. by M. Schoenauer et al. (Springer, Berlin, 2000), pp. 201–210
39. S. Luke, Code growth is not caused by introns, in *Late Breaking Papers at GECCO-2000* (2000), pp. 228–235
40. S. Luke, Issues in scaling genetic programming: breeding strategies, tree generation, and code bloat. PhD thesis, Department of Computer Science, University of Maryland (2000)
41. S. Luke, G.C. Balan, L. Panait, Population implosion in genetic programming, in *Proceedings of GECCO-2003*, ed. by E. Cantú-Paz et al. (Springer, Berlin, 2003), pp. 1729–1739
42. S. Luke, Modification point depth and genome growth in genetic programming. Evol. Comput. **11**(1), 67–106 (2003)
43. S. Luke, L. Panait, Fighting bloat with nonparametric parsimony pressure, in *Proceedings of PPSN-2002*, ed. by J.M. Guervos et al. (Springer, Berlin, 2002), pp. 411–420
44. S. Luke, L. Panait, Lexicographic parsimony pressure, in *Proceedings of GECCO-2002*, ed. by W.B. Langdon et al. (Morgan Kaufmann, San Francisco, CA, 2002), pp. 829–836
45. S. Luke, L. Panait, A comparison of bloat control methods for genetic programming. Evol. Comput. **14**(3), 309–344 (2006)
46. N.F. McPhee, J.D. Miller, Accurate replication in genetic programming, in *Proceedings of ICGA'95*, ed. by L. Eshelman (Morgan Kaufmann, San Francisco, CA, 1995), pp. 303–309
47. N.F. McPhee, A. Jarvis, E.F. Crane, On the strength of size limits in linear genetic programming, in *Proceedings of GECCO-2004*, ed. by K. Deb et al. (Springer, Berlin, 2004), pp. 593–604
48. N.F. McPhee, R. Poli, A schema theory analysis of the evolution of size in genetic programming with linear representations, in *Proceedings of EuroGP-2001*, ed. by J. Miller et al. (Springer, Berlin, 2001), pp. 108–125
49. P. Nordin, W. Banzhaf, Complexity compression and evolution, in *Proceedings of ICGA'95*, ed. by L. Eshelman (Morgan Kaufmann, San Francisco, CA, 1995), pp. 318–325
50. P. Nordin, F. Francone, W. Banzhaf, Explicitly defined introns and destructive crossover in genetic programming, in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, ed. by J.P. Rosca (1995), pp. 6–22
51. P. Nordin, F. Francone, W. Banzhaf, Explicitly defined introns and destructive crossover in genetic programming, in *Advances in Genetic Programming 2*, ed. by P.J. Angeline, K.E. Kinnear Jr. (MIT Press, Cambridge, MA, 1996), pp. 111–134
52. U.-M. O'Reilly, F. Oppacher, Hybridized crossover-based search techniques for program discovery, in *Proceedings of the 1995 World Conference on Evolutionary Computation* (IEEE Press, Piscataway, NJ, 1995), pp. 573–578
53. R. Poli, General schema theory for genetic programming with subtree-swapping crossover, in *Proceedings of EuroGP-2001*, ed. by J. Miller et al. (Springer, Berlin, 2001), pp. 143–159

54. R. Poli, A simple but theoretically-motivated method to control bloat in genetic programming, in *Proceedings of EuroGP-2003*, ed. by C. Ryan et al. (Springer, Berlin, 2003), pp. 200–210
55. R. Poli, W.B. Langdon, S. Dignum, On the limiting distribution of program sizes in tree-based genetic programming, in *Proceedings of EuroGP-2007*, ed. by M. Ebner et al. (Springer, Berlin, 2007), pp. 193–204
56. R. Poli, N.F. McPhee, L. Vanneschi, The impact of population size on code growth in GP: analysis and empirical validation, in *Proceedings of GECCO-2008*, ed. by M. Keijzer et al. (ACM Press, New York, NY, 2008), pp. 1275–1282
57. R. Poli, N.F. McPhee, L. Vanneschi, Elitism reduces bloat in genetic programming, in *Proceedings of GECCO-2008*, ed. by M. Keijzer et al. (ACM Press, New York, NY, 2008), pp. 1343–1344
58. R. Poli, N.F. McPhee, L. Vanneschi, Analysis of the effects of elitism on bloat in linear and tree-based genetic programming, in *Genetic Programming Theory and Practice VI*, ed. by R. Riolo et al. (Springer, Berlin, 2008), pp. 91–111
59. D. Rochat, Programmation génétique parallèle: opérateurs génétiques variés et populations dynamiques. MSc thesis, Université de Lausanne, Université de Genève (2004)
60. D. Rochat, M. Tomassini, L. Vanneschi, Dynamic size populations in distributed genetic programming, in *Proceedings of EuroGP-2005*, ed. by M. Keijzer et al. (Springer, Berlin, 2005), pp. 50–61
61. J.P. Rosca, Generality versus size in genetic programming, in *Proceedings of GP'96*, ed. by J.R. Koza et al. (MIT Press, Cambridge, MA, 1996), pp. 381–387
62. J.P. Rosca, Analysis of complexity drift in genetic programming, in *Proceedings of GP'97*, ed. by J.R. Koza et al. (Morgan Kaufmann, San Francisco, CA, 1997), pp. 286–294
63. S. Silva, Controlling bloat: individual and population based approaches in genetic programming. PhD thesis, Departamento de Engenharia Informatica, Universidade de Coimbra (2008)
64. S. Silva, J. Almeida, GPLAB—a genetic programming toolbox for MATLAB, in *Proceedings of the Nordic MATLAB Conference*, ed. by L. Gregersen (2003), pp. 273–278
65. S. Silva, J. Almeida, Dynamic maximum tree depth—a simple technique for avoiding bloat in tree-based GP, in *Proceedings of GECCO-2003*, ed. by E. Cantú-Paz et al. (Springer, Berlin, 2003), pp. 1776–1787
66. S. Silva, E. Costa, Dynamic limits for bloat control—variations on size and depth, in *Proceedings of GECCO-2004*, ed. by K. Deb et al. (Springer, Berlin, 2004), pp. 666–677
67. S. Silva, P.J.N. Silva, E. Costa, Resource-limited genetic programming: replacing tree depth limits, in *Proceedings of ICANNGA-2005*, ed. by B. Ribeiro et al. (Springer, Berlin, 2005), pp. 243–246
68. S. Silva, E. Costa, Resource-limited genetic programming: the dynamic approach, in *Proceedings of GECCO-2005*, ed. by H.-G. Beyer et al. (ACM Press, New York, NY, 2005), pp. 1673–1680
69. S. Silva, E. Costa, Comparing tree depth-limits and resource-limited GP, in *Proceedings of CEC-2005*, ed. by D. Corne et al. (IEEE Press, Piscataway, NJ, 2005), pp. 920–927
70. P.W.H. Smith, K. Harries, Code growth, explicitly defined introns, and alternative selection schemes. Evol. Comput. **6**(4), 339–360 (1998)
71. T. Soule, J.A. Foster, Removal bias: a new cause of code growth in tree based evolutionary programming, in *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation* (IEEE Press, Piscataway, NJ, 1998), pp. 781–786
72. T. Soule, Code growth in genetic programming. PhD thesis, College of Graduate Studies, University of Idaho (1998)
73. T. Soule, J. Foster, Code size and depth flows in genetic programming, in *Proceedings of GP'97*, ed. by J. Koza et al. (Morgan Kaufmann, San Francisco, CA, 1997), pp. 313–320
74. T. Soule, R.B. Heckendorn, An analysis of the causes of code growth in genetic programming. Genet. Program. Evolvable Mach. **3**(1), 283–309 (2002)
75. J. Stevens, R.B. Heckendorn, T. Soule, Exploiting disruption aversion to control code bloat, in *Proceedings of GECCO-2005*, ed. by H.-G. Beyer et al. (ACM Press, New York, NY, 2005), pp. 1605–1612
76. M.J. Streeter, The root causes of code growth in genetic programming, in *Proceedings of EuroGP-2003*, ed. by C. Ryan et al. (Springer, Berlin, 2003), pp. 443–454
77. W.A. Tackett, Recombination, selection, and the genetic construction of genetic programs. PhD thesis, Department of Electrical Engineering Systems, University of Southern California (1994)
78. M. Tomassini, L. Vanneschi, J. Cuendet, F. Fernandez, A new technique for dynamic size populations in genetic programming, in *Proceedings of CEC-2004* (IEEE Press, Piscataway, NJ, 2004), pp. 486–493

79. T. Van Belle, D.H. Ackley, Uniform subtree mutation, in *Proceedings of EuroGP-2002*, ed. by J.A. Foster et al. (Springer, Berlin, 2002), pp. 152–161
80. L. Vanneschi, Theory and practice for efficient genetic programming. PhD thesis, Faculty of Sciences, University of Lausanne (2004)
81. B.-T. Zhang, H. Mühlenbein, Balancing accuracy and parsimony in genetic programming. Evol. Comput. **3**(1), 17–38 (1995)