Bloat: Past, Present, Future

Sara Silva\textsuperscript{1,2}  Stephen Dignum\textsuperscript{3}  Leonardo Vanneschi\textsuperscript{4,1}

\textsuperscript{1} INESC-ID Lisboa  
Rua Alves Redol 9  
1000-029 Lisboa  
Portugal

\textsuperscript{2} Center for Informatics and Systems of the University of Coimbra
Portugal

\textsuperscript{3} School of Computer Science and Electronic Engineering
University of Essex
UK

\textsuperscript{4} Dipartimento di Informatica, Sistemistica e Comunicazione (D.I.S.Co.)
University of Milano-Bicocca
Italy

Contents

Abstract

1 Introduction

2 Past bloat theories
   2.1 Hitchhiking
   2.2 Defense Against Crossover
   2.3 Removal Bias
   2.4 Fitness Causes Bloat
   2.5 Modification Point Depth

3 Bloat Control Methods
   3.1 Evaluation
   3.2 Selection
   3.3 Breeding
   3.4 Survival
   3.5 Others

4 Crossover Bias
   4.1 Program Size Distributions and Bloat
   4.2 Operator Length Bias

5 Operator Equalisation
   5.1 Basics
   5.2 Variants
      5.2.1 Original Implementation
      5.2.2 DynOpEq – Dynamic Operator Equalisation
      5.2.3 MutOpEq – Mutation-Based Operator Equalisation
   5.3 Operator Equalisation in Practice
      5.3.1 Calculating the Target Distribution
      5.3.2 Following the Target Distribution
6 Benchmark Results
6.1 Experiments .............................................. 23
6.2 Results .................................................. 24
   6.2.1 Optimistic View .................................... 24
   6.2.2 Pessimistic View .................................... 24
   6.2.3 Realistic View ...................................... 25
7 Real World Results
7.1 Drug Discovery ........................................... 28
   7.1.1 Experiments ......................................... 28
   7.1.2 Results for Bioavailability .......................... 29
   7.1.3 Results for Toxicity .................................. 30
7.2 Classification of Satellite Images ........................ 31
   7.2.1 Experiments ......................................... 31
   7.2.2 Results .............................................. 31
8 Discussion
8.1 MutOpEq versus DynOpEq ............................... 33
8.2 Bloat and Overfitting .................................... 34
8.3 Dynamics of DynOpEq .................................... 35
   8.3.1 Rejections and Target Distributions ................. 35
   8.3.2 Rejections and Function Sets ....................... 38
9 The Future
9.1 Improvements ............................................ 40
9.2 Extensions ............................................... 41
9.3 Final Considerations ................................... 42
Acknowledgments .............................................. 43
Bibliography .................................................. 44
Abstract

Bloat can be defined as an excess of code growth without a corresponding improvement in fitness. This problem has been one of the most intensively studied subjects since the beginnings of Genetic Programming. This report presents a deep analysis of the past and present contributions dealing with the problem of bloat, and discusses the prospects of future challenges and developments. In particular, it reviews all the main theories explaining bloat, and many of the bloat control methods available so far, with particular attention to the new Crossover Bias theory and the bloat control method it inspired, Operator Equalisation. Although still recent and requiring improvements, Operator Equalisation has already proven to be more than just a bloat control method. It reveals novel evolutionary dynamics that allow a successful search without code growth, and shows great potential to be extended and integrated into several different elements of the evolutionary process. The results presented clearly show that Genetic Programming using Operator Equalisation is essentially bloat free. After so many years of research, the true reasons for bloat are finally understood, and the path to its full extinction is clearly seen.
Chapter 1

Introduction

When Koza published the first book on Genetic Programming (GP) [50], most of the evolved programs therein contained pieces of code that did not contribute to the solution and could be 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 [6]. 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 rejoices at 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” [6].

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 [79,118,119]. Compact solutions are thought to be more robust and generalize better [48,79,101,126,133], although this has been empirically questioned in recent work [114,115]. Introns also seem to provide some protection against the destructive effects of crossover and other genetic operators [3,18,79,116,118] although this may not always be helpful. The usage of explicitly defined artificial introns has yielded generally good results in linear GP [65,81,82], but in tree-based GP it usually degraded the performance of the search process [5,16,116].

Regardless of the possible benefits that redundant code may provide to GP, its excessive proliferation usually causes serious side effects. Computational resources may be totally exhausted in the storage, evaluation and swapping of code that con-
tributes nothing to the final solution, preventing GP from performing the effective search needed to find better solutions. Also, redundant code may be very difficult to eliminate from the final solution provided by GP, thus compromising its understandability and usefulness to a human user. Although lacking a formal definition, this problem is known as bloat and can be recognized as an excess of code growth without a corresponding improvement in fitness. It 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 [14, 54, 60, 62, 64].

Bloat has been one of the most intensively studied subjects since the beginnings of GP, and already subject to different theoretic and analytic studies [58, 65, 76, 87, 88, 101, 102]. Several theories concerning why bloat occurs have been advanced, and numerous bloat control methods have been proposed. In recent years some new methods have proven to be very successful in a varied set of benchmark problems, like Double Tournament [72] and Dynamic Limits [111], but until now a definitive solution was yet to be found. It was the publication of a new theory explaining the emergence of bloat, the Crossover Bias theory [26, 92], that caused a renewed interest in the problem. Firmly based on the Crossover Bias theory, a new bloat control method called Operator Equalisation [27, 112] has achieved outstanding results in several benchmark problems. In the last few months it has been tested in real world applications involving biomedical [114, 130] and multispectral image data [115]. The results so far reveal that, after so many years of intense bloat research, it is finally possible for GP to find simple and accurate solutions that are completely bloat free, and more likely to be “human readable” and effectively usable/useful in the real world.

This report presents a historical view of the research done in bloat since the beginnings of GP, and provides an updated picture of the work currently being done on this subject, and how it can lead to the effective and efficient extinction of bloat. The next chapter describes the main past theories concerning the reasons why bloat occurs. Chapter 3 introduces a taxonomy of most of the bloat control methods published until recently. Chapter 4 presents the Crossover Bias theory, while Chapter 5 describes the different variants of Operator Equalisation available so far. Chapter 6 summarizes the results achieved by Operator Equalisation in benchmark problems, while Chapter 7 reports some of the results obtained in real world problems. Chapter 8 provides a general interpretation of the results and discusses related issues. Finally, Chapter 9 offers many prospects of future research and concludes the report.
Chapter 2

Past bloat theories

This chapter describes the five main past 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 given along the years are not necessarily contradictory. Some appear to be generalizations or refinements of others, and several most certainly complement each other. They were all published between 1994 and 2003, with further discussion extending until 2006. Here 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 [126]. 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 Section 2.2) as the explanation for intron multiplication, based on the fact that the usage of brood recombination [3], a less destructive recombination strategy, did not result in less code growth [126]. An additional hypothesis for code growth, advanced by Altenberg and somewhat related to the removal bias theory later advanced by Soule (Section 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 [4]. 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 [126]. These results have later been reinforced by other experiments showing the absence of bloat when selection is random [13, 60, 65].
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 [3,18,74,79,116,118], also stated and referred to as the replication accuracy theory [74,88], intron theory [36,37,125] and protection theory [20]. It is based on the fact that standard crossover is usually very destructive [12, Chap. 6], [79, 81, 82]. 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 [79] and may not be completely applicable to tree-based GP [66, 67, 69, 101]. More specifically, introns can be roughly divided in two categories: inviable code and unoptimized code (or syntactic/structural and semantic introns [8, 20]). 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 [69]. 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 [83, 119] 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 [69, 70]. Additional experiments by Soule and Heckendorn using single node mutations have however suggested that code growth does occur in response to destructive operators [122].

Luke indeed rejects the defense theory in the context of tree-based GP [69] by using a simple procedure called marking [18]. 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 [20].
2.3 Removal Bias

Although presenting evidence to support the theory of defense against crossover (Section 2.2), Soule performed additional experiments with another non-destructive but less “rigorous” hill-climbing crossover [117, 118]. While the previous crossover retained only the offspring that were strictly better than their parents [119], 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 [64, 117, 118].

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 Section 2.1).

When using the more rigorous non-destructive crossover that only allows offspring with better fitness than their parents (Section 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 [69, 70]. When using the more rigorous crossover, the improvement of mean population fitness is indeed slower than when using the less rigorous crossover [117]. 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 [117], 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 [122].

2.4 Fitness Causes Bloat

The first theory that does not make introns responsible for bloat was advanced by Langdon and Poli [54, 60, 62, 64]. Also called solution distribution [118], diffusion theory [66, 67, 125], drift [20, 122], nature of search spaces [88] and entropy random walk [65], it has recently been identified simply by its main claim, fitness causes bloat [72]. 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 [63, 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 [126], experiments have shown that code growth does not occur when using random selection [13, 60, 65], not even when standard mutation is the only genetic operator [62]. 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 [38]. 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 [133], and Gelly et al., have also found evidence to support the claim that fitness causes bloat [36, 37].

2.5 Modification Point Depth

Another explanation for bloat in tree-based GP was advanced by Luke [66, 67, 69]. It has been called depth-correlation theory [125], but can also be referred to as depth-based theory or simply modification point depth [69].

Confirming previous results [44], 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 (Section 2.3). This may be regarded as a generalization of the original removal bias theory [122].

Luke denies that introns cause bloat [66]. 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 [67, 69, 72]. Streeter also suggests that code growth may be related to a measure of resilience, where resilience is directly related to tree size [125].
Chapter 3

Bloat Control Methods

Bloat control methods are so numerous and varied that it is hard to define a taxonomy to classify them, let alone enumerate them all. Because different control measures act at different stages of the evolutionary process, the following taxonomy reflects precisely that: where in the iterative GP process does the method apply. Evaluation, selection, breeding and survival are the four evolutionary stages considered. Given the extreme diversity of existent bloat control methods, and to avoid a further discretization of the evolution, some methods are simply included in a heterogeneous group called others. The following list includes bloat control methods published between 1993 and nowadays. Although extensive, it is probably not exhaustive. Other taxonomies can be found in the literature [63, Sect. 11.6], [71, 72, 134, 135].

3.1 Evaluation

Applying bloat control at the level of fitness evaluation is a very common practice, even in other evolutionary computation paradigms besides GP. Like all techniques where the size of an individual affects its probability of being selected for reproduction, the following belong to the wide family of parsimony pressure methods.

In parametric parsimony pressure, the fitness of each individual is a function of its raw fitness and its size, penalizing larger individuals. It has been used in GP since the beginnings of research into the discipline, also under different alternative names and variants like Occam’s razor and Minimum Description Length [17, 42, 48, 50, 121, 133]. Parsimony pressure is usually linear and constant during the evolution, but some techniques apply adaptive pressure where the adjustment of the fitness penalty varies during the evolution [133]. A major difficulty of all parametric methods is precisely its dependency on the correct setting of the parsimony coefficient, worsened by the fact that what is correct in the beginning of the evolutionary process may actually handicap the search later in the evolution. The effects of parametric parsimony pressure have been studied, and it was found that although it may speed the evolution, it may also cause the search process to converge on local optima [101–103, 120].

Different parsimony pressure methods have also been used, like the tarpeian method.
[88], where a fraction of individuals with above-average size are periodically “killed” by giving them such an extreme fitness value that effectively prevents them from being selected for reproduction. Notably one of the few theoretically-motivated bloat control techniques, the tarpeian method has not however been able to beat some older and newer techniques [72].

Most recently, a very efficient way of setting the parsimony coefficient dynamically during the run has been developed for parametric parsimony pressure methods. Based on solid theory, the method achieves complete control over the evolution of the average size of the individuals in the population, even forcing its shrinkage if necessary [93].

3.2 Selection

Unlike parametric parsimony, pareto-based parsimony does not modify the fitness of the individuals. Instead, parsimony is applied by selecting based on two objectives: fitness and size. Multi-objective parsimony methods do not introduce any parameters to the search process, but unfortunately their results have not been consistently good [15, 23, 24, 32, 59, 72, 85].

Other bloat control methods that act at the selection level include several types of tournament. Lexicographic parsimony pressure [71, 72] uses a tournament that always selects the smaller contestant from among individuals having the same fitness. In double tournament [72, 85] the contestants are already the winners of a previous tournament, the first based on size and the second based on fitness, or vice versa. In proportional tournament [72, 85] a proportion of tournaments will select winners based on size instead of fitness. Double tournament has recently proven to be one of the best bloat control methods available [72].

Pygmies and civil servants [105] always draws one parent from the pygmy list (short individuals) and another from the civil servant list (fit individuals), aiming at the creation of offspring that are both short and fit.

3.3 Breeding

Bloat control at the breeding level is performed with specific genetic operators that attempt to restrict code growth. Many different non-standard operators have been advanced, most of them searching for better performance [12, Chap. 6], but some specifically looking to end bloat.

Brood recombination (Section 2.1), also called soft brood selection and greedy recombination [3, 126], is a crossover operator that creates many more offspring than needed, by selecting different crossover points in both parents. From all the individuals created in each crossover, only the best become full-fledged offspring and candidate to the new generation. The number of effective offspring returned by this operator is always the same regardless of the size of the brood. The role of a large brood is to reduce the crossover destructiveness.

One-point crossover [89, 90], very similar to context preserving crossover [25], chooses crossover points that are common to both parents. Because parents are seldom
identical in size and shape, crossover points are restricted to parental structurally identical regions (in terms of function arity from the root node), usually meaning that relatively large subtrees are swapped. A variant named strict one-point crossover restricts identical regions to having both the same arity and the exact same functions. Less restrictive than one-point crossover, the same depths crossover [116] begins by choosing a random depth in the least deep parent, and then selects random crossover points at this depth, in both parents. This also increases the chances that larger subtrees are swapped. The popular 10/90% crossover that chooses non-terminals with 90% probability [50] is another attempt at swapping larger branches. Uniform crossover [91] is similar to one-point crossover in that swapping of genetic material can only be done within common parental regions, and it also includes a strict variant. The difference is that in uniform crossover only single nodes are swapped, not entire branches (except when the swapped node is a non-terminal at the boundary of the common region). Smooth uniform crossover (and the related smooth point mutation) [84] is a variant that does not swap entire nodes and instead interpolates their behavior to allow smaller movements around the solution space. One-point, same depth and uniform crossovers all have one thing in common: they maintain the depth of the offspring within the limits established by the initial population.

Size fair crossover [56, 57] behaves much like standard tree crossover, except in regard to the choice of the crossover point in the second parent. A bound is placed on the amount of genetic material exchanged in a single operation, so the selected branch on the second parent must not exceed a certain size limit related to the size of the selected branch on the first parent. Size fair mutation [54, 64] behaves in a similar manner regarding the size of the newly created subtree. Based on fair crossover, homologous crossover [56] attempts at preserving the context in which the subtrees are swapped, by selecting the second branch as similar as possible to the branch in the first parent. The aligned homologous crossover [80] and the maximum homologous crossover [86] are homologous crossovers specifically developed for linear GP. Also for linear GP, page-based crossover [40] attempts to reduce bloat by exchanging fixed size blocks so that individuals do not grow larger than the largest individual in the population.

Other bloat control operators include a crossover that truncates excess depth [73], and also specialized genetic operators that depend on the size or depth of the parents: large trees are modified with operators likely to reduce their size, while smaller trees are likely to grow [47].

Two techniques where the number of operations performed on a single (pair of) parent(s) is dependent on the size of the breeding trees. Uniform subtree mutation [128] applies several rounds of standard mutation to the same individual. The larger the parent tree, the higher the number of operations. Following the same rationale, multiple crossovers have also been used as an attempt to break the tree resilience that is correlated to code growth [124] (Section 2.5).

In [1] a new bloat control method, called prune and plant, has recently been proposed. It is inspired by the homonymous strategy used in agriculture. It is used mainly for fruit trees and it consist of pruning some branches of trees and planting them in order to grow new trees. The idea is that the worst tree (in terms of fitness) in a population will be substituted by branches pruned from one of the best trees and planted in its
place. This way the offspring trees will be of smaller size than the ancestors, effectively reducing bloat. This method has been implemented modifying the crossover operator. It has been shown to work well for the bankruptcy prediction problem and for several other benchmark problems.

### 3.4 Survival

Bloat control at the survival level can be applied on an individual basis, where each candidate to the new generation must conform to some standards if it is to become an effective population member, or on a population basis, where the population as a whole must obey some restrictions, regardless of the particular characteristics of each individual.

Some of the individual-based techniques are usually described as specific crossover operators, and as such could be classified as breeding restrictions (Section 3.3). But this really depends on the implementation details, and it is best to keep things separate. Any genetic operator can be used along with these bloat control techniques, intact and unrestricted. Only after the breeding process is finished are the new individuals filtered as part of the survival process.

#### Individual Based

The first bloat control method ever used on tree-based GP is still the most popular and commonly used. Traditionally it imposes a fixed tree depth limit on the individuals accepted into the population. When a genetic operator creates an offspring that violates this limit, one of its parents is chosen for the new generation instead [50]. Alternatively, the operator can be retried until it produces a valid individual [73], or the invalid individuals accepted but given null fitness [75]. Size limits have been used instead of depth limits, where size is the number of tree nodes [52, 58, 60, 64]. However, 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 [63, Chap. 10], [35, 61]. Another study deals with the impact of size limits on the average size of the individuals in the population [75], and recent work related to the Crossover Bias theory reports that size limits actually speed code growth in the early stages of the run [28] (see Section 4.1).

**Dynamic Maximum Tree Depth** [107, 111] is a bloat control technique inspired in the traditional tree depth limit. It also imposes a depth limit on the individuals accepted into the population, but this one is dynamic and able to increase during the run. Variants of this technique include the implementation of a heavy limit that can also decrease during the run, and the usage of a limit on size instead of depth [108]. Dynamic Maximum Tree Depth and its variants are collectively called Dynamic Limits.

Another technique that replaces offspring with their parents when restrictions are not respected is the *hill-climbing crossover* [117–119] (Section 2.2), also called *pseudo-hill-climbing* [70, 72]. In this technique, only the individuals that are not worse than their parents, or are strictly better than their parents, are allowed to enter the population. Individuals who do not conform to these standards are replaced by their parents. This technique counteracts the natural destructiveness of crossover. An identical technique is called *improved fitness selection*, while *changed fitness selection* only accepts in-
individuals with a different fitness from their parents, better or worse [116]. A similar approach at diversity pressure has been used in a different manner, much like parametric parsimony pressure (Section 3.1), by including a penalty on the fitness of the offspring that have the same fitness as their parents [55].

Developed alongside the Crossover Bias theory (to be described in Chapter 4), a new method for bloat control has been proposed, called operator equalisation [27]. It is capable of accurately controlling the distribution of sizes inside the population by probabilistically accepting each individual based on its size, where the probabilities are calculated considering the target distribution. This method will be thoroughly described in Chapter 5.

Population Based The first attempt to control bloat using restrictions at the population level was made with the implementation of a fixed limit on the total number of nodes of the entire population [132]. This idea was further developed and tree nodes were regarded as the natural resources that individuals need to survive [113]. Its hybridization with Dynamic Limits resulted in the implementation of a dynamic limit on the amount of resources the population can use [109, 110], whose variations depend on the evolution of mean population fitness. The concept of natural resources in GP was named Resource-Limited GP. The introduction of limits at the population level results in automatic population resizing.

Simpler approaches like the systematic shrinking of the population have also been used in order to save computational effort and thus counter the effects of bloat [33, 34, 68]. Another, more sophisticated variable population approach, is implemented by explicitly inserting or deleting individuals depending on how the best fitness is evolving [129, Sect. 7.1] [21, 99, 100, 127]. Individuals are suppressed as long as the best individual in the population keeps improving, and new individuals are added when the best fitness stagnates.

3.5 Others

Some bloat control methods do not really fit into any of the previous categories, like the waiting room and death by size [72, 85]. The waiting room implements a queue where newly created individuals must wait until they can enter the population. The larger the individual, the longer it must wait. Death by size is a technique designed specifically for steady-state GP, as opposed to generational GP. At each time step some individuals are selected to be removed from the population and replaced by the new children. Larger individuals are more likely to be removed.

Code editing is simply the removal of redundant code, both in tree-based and in linear GP. It has been performed since early times to clean and simplify the final solution [50], and has also been used along the evolutionary process as an attempt to counteract bloat. Code editing can be done before [19] or after [43] evaluating an individual, with a mutation operator that simplifies [31] or a crossover operator that deletes redundant regions [16] or through any other simplification system that acts periodically on the individuals [41, 121]. However, it has been shown that code editing can lead to premature convergence [39].
In [49] node-based numerical simplification has recently been proposed as a tree pruning criterion to control program size. Even if this method is not based on a rigorous theory explaining bloat, the authors show that simplification results in reductions in expected program size, memory use and computation time. Furthermore, they show that numerical simplification performs at least as well as algebraic simplification alone, and in some cases outperforms algebraic simplification itself. These results are purely experimental, and have been tested on a relatively small set of test problems, but they shed a light on the complex and in many ways still unexplored relationship between bloat and program complexity.

In the same vein, a new code simplification method for symbolic regression problems called \textit{equivalent decision simplification} has been presented in [78]. In this method, subtrees are evaluated over the set of regression points and classified in equivalence groups (where two subtrees are defined equivalent if they return the same outputs for the same sets of inputs). After that, the subtrees in an equivalence class are replaced with the smallest ones (in terms of number of nodes) in that class. The effectiveness of this method is experimentally shown.

\textit{Explicitly defined introns} are intended as substitutes of naturally occurring introns [7, 16, 65, 81, 82, 116]. This technique consists on the inclusion of special nodes with added functionality that adapt the likelihood of crossover or mutation to operate at specific locations within the code.

\textit{Dynamic fitness}, where the fitness measure is based on co-evolution, or calculated on a variable set of fitness cases, has been extensively used as an attempt to improve the convergence ability of GP [53, Sect. 2.4.2], and only rarely as a bloat control technique [55].

Although not initially developed as a bloat control method, promoting the modularization and reusability of GP structures often results in more parsimonious code. Some modularization techniques are the creation of Automatically Defined Functions [51] and Automatically Defined Macros [123], Module Acquisition [9] and Adaptive Representation Learning [104].

Finally, some less common forms of GP seem to have the ability of reducing code growth, like \textit{relaxed GP} [22], or do not seem to be affected by bloat at all, like \textit{stochastic grammar-based GP} [97] and \textit{cartesian GP} [77]. In relaxed GP, the values of the desired solution are relaxed so that any value within a specified interval will be considered correct. Stochastic grammar-based GP is a grammar-based GP framework where the information is stored as a probability distribution on the grammar rules, rather than in a population. In cartesian GP, a program is represented as an indexed graph.

An attempt to propose a theoretically motivated method to control bloat has recently been done in [2], where the use of new graph structural GP for automatic programming is discussed. This method creates finite state machines (FSM) by evolution. Generally, FSM must define their transition rules for all combinations of states and possible inputs, thus the FSM program will become large and complex when the number of states and inputs is large. In [2] the nodes are connected by trajectory information sets. According to the author, this enables only the essential behavior of the problem to be used in the network flow, so simplifying the proposed solutions. Although interesting and rigorous, this method is defined for a complex and quite unusual form of GP (graph structural GP) and it is not easily extendable to standard tree based GP.
Chapter 4

Crossover Bias

The most recent theory concerning bloat is the Crossover Bias theory by Dignum and Poli [26, 28, 92, 94]. It explains code growth in tree-based GP by the effect that standard subtree swapping crossover has on the distribution of tree sizes in the population. This chapter first describes the theory in detail including its origins and a number of implications for experimental design. It then argues that this form of bloat will affect any GP experimentation with variation operators that regularly sample relatively small programs, i.e., those that cannot achieve a fitness comparable to more sophisticated larger programs.

4.1 Program Size Distributions and Bloat

Whenever standard subtree swapping crossover is applied, the amount of genetic material removed from the first parent is, on average, the exact same amount inserted in the second parent, and vice versa. The mean tree size therefore remains unchanged. However, as the population undergoes repeated crossover operations, it approaches a particular limiting distribution of tree sizes (it has been hypothesized and experimentally demonstrated that this distribution is a Lagrange distribution of the second kind [45, 46, 92]), where small individuals are much more frequent than the larger ones (see Fig. 4.1 as illustration). For example, crossover generates a high amount of single-node individuals. For all but the very simplest of problems, very small individuals will generally be unfit. Selection tends, therefore, 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 [26].

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 [26]. Indeed, the theory was created in light of an investigation into the creation of
Figure 4.1: Comparison between empirical size/length distributions and ‘arity histogram’ based Lagrange models created with arity 1, 2, 1&3, and 1-4 functions and terminals only, (a) to (d) respectively. All initialized with Full method [50] (depth = 15, 3, 3, 3, initial mean size $\mu_0 = 16.00$, 15.00, 15.00, 25.38, mean size after 500 generations $\mu_{500} = 16.15$, 14.19, 15.75, 23.72 respectively). Standard Subtree swapping crossover is applied with uniform selection of crossover points on a flat fitness landscape. Population size = 100,000. Results were averaged over 20 runs. Taken from [29]
a crossover “friendly” initialization method - during experimentation a bloating effect was observed to take place as initial populations converged to a Lagrange distribution.

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 [28]. Along with further theoretical developments, it has also been shown that smaller populations bloat more slowly [94], and that elitism reduces bloat [95, 96].

4.2 Operator Length Bias

One can argue that any variation operator that has a bias towards smaller programs will cause bloat in this way and the theory should be renamed to operator length bias. With this in mind, it is relatively straightforward to conduct experimentation to recreate the length bias bloating effect by altering sampling bias of small programs during a GP run. To illustrate, in Fig. 4.2, using standard problems from the ECJ evolutionary computation toolkit\(^1\), we show the effect of applying a minimum child size (MCS) to crossover, i.e., potential children are rejected if they were smaller than a specific size. We use size threshold values of 1 (no rejections), 2, 4, 8, 16, 32, and 64. Note, as the minimum size gets sufficiently large, most initial individuals are too small to create valid children and so after 1,000 failed tries MCS results in an ordinary crossover. As we can see there is an increase in average program length for the acceptance of shorter programs by the crossover operation for both Artificial Ant and Symbolic Regression problems, which is not supported by corresponding best fitness values, i.e., programs in these populations tend to get larger but generally do not find fitter solutions compared to the other populations.

On further inspection one can see that removing programs shorter than 8 nodes works well for the Artificial Ant problem whilst performs relatively badly for the Symbolic Regression problem. The idea of automatically selecting programs of certain lengths, with a view to improving fitness, is discussed in the next chapter.

\(^1\)ECJ – Evolutionary Computation in Java, http://cs.gmu.edu/~eclab/projects/ecj/
Figure 4.2: Comparison between minimum child size (MCS) constraints for Artificial Ant (using Santa Fe Trail) and Symbolic Regression (to fit $x^4 + x^3 + x^2 + x$) problems. Average lengths are shown in (a) & (b), and average best fitness (lower is better) in (c) & (d). Experimental parameters as for ECJ standard, taken from [50], except uniform selection of crossover points is applied and depth limits are removed. No mutation is employed and population size = 1024. Results were averaged over 100 runs.
Chapter 5

Operator Equalisation

Some of the most popular methods to control code growth rely on the usage of length limits to prevent the larger individuals from entering the population (see Chapter 3). Although this may effectively keep the population from growing beyond the fitness convergence limit [63], it was found that length limits actually speed code growth in the beginning of the run because they promote the proliferation of small unfit individuals [28]. Therefore, and counterintuitively, to avoid code growth we must also prevent the smaller individuals from entering the population. Developed alongside the Crossover Bias theory [26, 28, 92, 94], Operator Equalisation (OpEq) is a recent bloat control technique that allows an accurate control of the program length distribution during a GP run. This chapter describes the original idea along with its two improved variants, and explains how to use it in practice.

5.1 Basics

To explain how OpEq works, we use the concept of a histogram. Each bar of the histogram can be imagined as a bin containing those programs whose length (i.e., total number of nodes) is within a certain interval. The width of the bar determines the range of lengths that fall into this bin, and the height specifies the number of programs allowed within. The former is called bin width and the latter bin capacity. All bins are the same width, placed adjacently with no overlapping. Each length value, \( l \), belongs to one and only one bin \( b \), identified as follows:

\[
 b = \left\lfloor \frac{l - 1}{\text{bin\_width}} \right\rfloor + 1 \tag{5.1}
\]

For instance, if \( \text{bin\_width} = 5 \), bin 1 will hold programs of lengths 1...5, bin 2 will hold programs of lengths 6...10, etc. The set of bins represents the distribution of program lengths in the population. OpEq biases the population towards a desired target distribution by accepting or rejecting each newly created individual into the population (and into its corresponding bin).
5.2 Variants

5.2.1 Original Implementation

The original implementation of OpEq controls the distribution of sizes inside the population by probabilistically accepting each newly created individual based on its size. The probabilities of acceptance are calculated considering a predetermined target distribution. A maximum allowed program length, and the desired number of length classes, or bins, are used to calculate the width of the bins. Except for the probabilities of acceptance, all these elements remain static throughout the run.

Initially, the probability of acceptance for each bin is calculated to be directly proportional to the desired frequency of individuals on the corresponding target bin. After each generation the current distribution is used to update the probabilities of acceptance so that the target is approximated at a certain rate (see [27] for details). Naturally, the probabilistic nature of the acceptance procedure allows large discrepancies between the target and current distributions. For example, if crossover presents large numbers of programs of a certain size in one generation the corresponding bins will overflow in comparison with the target. This is, however, corrected with a punitive acceptance probability in the next generation.

The original OpEq technique was tested on a 10-variate symbolic regression problem, and on the 10-bit even parity problem, using different target distributions. All the targets were easily reached, and some distributions proved more efficient for one or the other problem, but none provided the best results in both problems (see [27] for details).

5.2.2 DynOpEq – Dynamic Operator Equalisation

The original OpEq implementation, where the user was required to specify the target distribution and maximum program length, rapidly evolved to a self adapting implementation where both these elements are automatically set and dynamically updated to provide the best setting for each stage of the evolutionary process. This variant is called DynOpEq [112, 114].

There are two tasks involved in DynOpEq: calculating the target (in practical terms, defining the capacity of each bin) and making the population follow it (making sure the individuals in the population fit the set of bins). The first task did not exist in the original OpEq implementation, while the second task is also different from the original OpEq. In DynOpEq, the acceptance or rejection of the newly created individuals is based not only on their length, but also on their fitness. Both tasks are explained in Section 5.3.

DynOpEq has been tested on four well-known GP benchmark problems (symbolic regression, artificial ant, even-parity and multiplexer) [112], as well as in different real world applications [114, 115, 130]. A summary of the results is presented in Chapters 6 and 7.
5.2.3 MutOpEq – Mutation-Based Operator Equalisation

Another variant of the original OpEq was developed, where both target distribution and maximum program length are also automatically set and dynamically updated during the run. The same two tasks (calculating and following the target) specified for DynOpEq are adopted, but the implementation of the second task differs from DynOpEq. Instead of rejecting individuals, this variant transforms them by slightly mutating their genotype until they reach the desired length. This variant is called MutOpEq. It has been tested in the same problems as DynOpEq, and the results are summarized in Chapters 6 and 7.

5.3 Operator Equalisation in Practice

5.3.1 Calculating the Target Distribution

Provided with a simple and effective method for biasing the search towards the desired program lengths, the question immediately arises: what is the best program length distribution? The first steps taken in this direction [27] revealed that (1) it depends on the problem and (2) it depends on the stage of the run. The search should be biased towards program lengths that have better fitness, so for calculating the dynamic target length distribution we simply follow fitness.

For each bin, the average fitness of the individuals within is calculated, and the target is directly proportional to these values. Bins with higher average fitness will have higher capacity, because that is where search is proving to be more successful. Formalizing, the capacity, or target number of individuals, for each bin $b$, is calculated as:

$$\text{bin\_capacity}_b = \text{round}(n \times (\bar{f}_b/\sum_i \bar{f}_i))$$

(5.2)

where $\bar{f}_i$ is the average fitness in the bin with index $i$, $\bar{f}_b$ is the average fitness of the individuals in $b$, and $n$ is the number of individuals in the population. The rounded capacity values may not actually add to the exact number of individuals in the population, but that is irrelevant because the target does not have to be exactly matched, as will be explained in Section 5.3.2.

Initially based on the first randomly created population, the target is updated at each generation, always based on the fitness measurements of the current population. This creates a fast moving bias towards the areas of the search space where the fittest programs are, avoiding the small unfit individuals resulting from the crossover bias, as well as the excessively large individuals that do not provide better fitness than the smaller ones already found. In short, the dynamic target is capable of self adapting to any problem and any stage of the run.

5.3.2 Following the Target Distribution

In OpEq, every newly created individual must be validated before eventually entering the population. The details differ between the two variants DynOpEq and MutOpEq. The main difference resides on the fact that DynOpEq rejects the individuals that do
not fit the target, and MutOpEq mutates the individuals until they fit the target. Fig. 5.1 shows the pseudocode of both implementations: the cycle that generates a new population, followed by the details of the procedure used to validate the new individuals.

The cycle that generates a new population is common to both implementations. It simply consists of the generation of new individuals until the number of accepted individuals \( n_{\text{accepted}} \) reaches the intended population size \( \text{pop}_{\text{size}} \). The procedure that validates each new individual also has many parts in common. In both implementations, the length of the new individual is measured, and its corresponding bin is identified using Equation 5.1. If this bin already exists and is not full (meaning that its capacity is higher than the number of individuals already there), the new individual is immediately accepted. If the bin still does not exist (meaning it lies outside the current target boundaries) the fitness of the individual is measured and, in case we are in the presence of the new best-of-run (the individual with best fitness found so far), the bin is created to accept the new individual, becoming immediately full. Any other non-existing bins between the new bin and the target boundaries also become available with capacity for only one individual each. The dynamic creation of new bins frees the OpEq techniques from the fixed maximum program length that was present in the very first implementation. The criterion of creating new bins whenever needed to accommodate the new best-of-run individual is inspired by a successful technique to control code growth called Dynamic Limits [111].

The differences between DynOpEq and MutOpEq are revealed when the intended bin exists but is already at its full capacity, or when the intended bin does not exist and the new individual is not the best-of-run. In the first case, DynOpEq evaluates the individual and, if we are in the presence of the new best-of-bin (meaning the individual has better fitness than any other already in that bin), the bin is forced to increase its capacity and accept the individual. Otherwise, the individual is rejected. Permitting the addition of individuals beyond the bin capacity allows a clever overriding of the target distribution, by further biasing the population towards the lengths where the search is having a higher degree of success. In the second case, when the bin does not exist and the individual is not the best-of-run, rejection always occurs. There is an obvious computational overhead in evaluating so many individuals that end up being rejected. This subject has been extensively addressed in previous work [112] where the main conclusion was that most rejections happen in the beginning of the run and refer to very small individuals.

MutOpEq, on the other hand, never rejects individuals. In both cases mentioned above, this new implementation performs the same action: it searches for the closest existing non-full bin, and then mutates the new individual until its length fits this bin. The mutations used are “weak” mutations in the sense that they modify the least possible amount of code each time they are applied. Loosely based on the shrink and grow mutations defined in [131], our weak mutations can be informally described as follows:

**Shrink** Chooses a terminal branch (a minimum-depth branch containing only one function whose arguments are all terminals) and replaces it with one of its terminal nodes (one of the arguments of the function);

**Grow** Chooses a terminal node and replaces it with a terminal branch where one of the arguments is the replaced terminal.
One or the other is used, depending on whether the need is to reduce or enlarge the individual. Whenever there is a tie (i.e., there are two existing non-full bins at the same distance), the preferred option is to reduce. The same individual will undergo several mutations until it reaches the desired length. It is expected that, by iteratively inserting or removing only terminal elements, instead of inserting or removing a larger branch in a single operation, the impact on the fitness of the individual is minimized [44, 69].
generation of a new population:

```plaintext
n_accepted = 0
while (n_accepted < pop_size) do
  select parents from population
  apply genetic operator
  for each child i:
    if validate(i) then
      insert i in the new population
      n_accepted = n_accepted + 1
    endif
  endfor
endwhile
```

validate(individual i):

```plaintext
l = length of individual i
b = bin that holds individuals of length l
if b exists
  if b is full
    validate = false
    if DynOpEq and i is the new best-of-bin
      validate = true
    endif
  else
    validate = true
  endif
else if i is the new best-of-run
  create new bin
else
  validate = false
endif
if MutOpEq and validate = false
  closest_b = non-full bin closest to b
  mutate i until it fits closest_b
  validate = true
endif
```

Figure 5.1: Pseudocode of both OpEq techniques (DynOpEq and MutOpEq): the cycle that generates a new population, followed by the details of the procedure used to validate the new individuals
Chapter 6

Benchmark Results

Here we present a summary of the experiments performed and the results obtained by OpEq in a particular set of benchmark problems widely used in the literature. They represent a varied selection in terms of bloat dynamics and responses to different bloat control techniques. A new technique can be regarded as successful when it provides better results than what had been achieved so far. But what are the criteria to decide whether some given results are better than others? We show how misleading some plots can be, by presenting an optimistic view, followed by a pessimistic view, of the results achieved. The differences are striking. Then we present what we believe is a fair and realistic view of the results. All the claims of significant differences in the results are supported by statistical evidence as described in Section 6.1 (Tools).

6.1 Experiments

Problems. Four different problems were chosen to test the OpEq techniques: Symbolic Regression of the quartic polynomial \(x^4 + x^3 + x^2 + x\), with 21 equidistant points in the interval \([-1, 1]\), Artificial Ant on the Santa Fe food trail, 5-Bit Even Parity, and 11-Bit Boolean Multiplexer. The function sets are the same as in [50]. The Regression problem uses no random constants. In all four problems, fitness was calculated such that lower values represent better fitness. For details see [112].

Techniques. Both DynOpEq and MutOpEq were tested, each using three different bin widths: 1, 5, 10. Each technique is identified by a name and number, where the name identifies the OpEq variant and the number specifies the bin width (e.g. DynOpEq1, MutOpEq5). These were compared with four other techniques: a no-limits approach were individuals grow completely free of restrictions (NoLimits); the popular Koza approach of limiting tree depth to the fixed value of 17 [50] (StdGP); the successful Dynamic Limits approach where the maximum allowed tree depth is initially set with a low value, and only increased when needed to accommodate a new best-of-run individual [106, 111] (DynDepth); and the recent successful Double Tournament where the selection for reproduction is based on two tournaments, one for size.
and one for fitness [72, 85] (Double). Like OpEq, neither DynDepth nor Double use any fixed maximum limits.

**Parameters.** 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. Tree initialization was performed with the Ramped Half-and-Half method with a maximum initial depth of 6 (the same value was used to initialize the dynamic limit of DynDepth). Selection for reproduction used tournaments of size 7 (the same value was used in Double). The reproduction (replication) rate was 0.1, no mutation was used, and crossover adopted a uniform selection of crossover points. Selection for survival was not elitist. For more details consult the settings in [112].

**Tools.** All the experiments were performed using a modified version of GPLAB\(^1\). Statistical significance of the null hypothesis of no difference was determined with pairwise Kruskal-Wallis non-parametric 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 shown next. The median is also more robust to outliers.

### 6.2 Results

#### 6.2.1 Optimistic View

Figure 6.1 shows the evolution of the best fitness during the run for all four problems. This is the most traditional way of presenting such results: fitness versus generations. Looking at the results presented this way, DynOpEq seems to be very successful indeed, for all three different bin widths. For all problems, DynOpEq is able to improve fitness faster and in most cases reaches significantly better values than the other techniques. MutOpEq, on the other hand, is among the worst.

However, this view of the results is extremely optimistic for DynOpEq. When describing how to make the population follow the target length distribution by filtering the individuals that enter the new generation (Section 5.3.2) it was mentioned that a major computational overhead was being introduced because of the need to evaluate the individuals before deciding whether to accept or reject them. In each generation of DynOpEq, many more individuals are created and evaluated than in the other techniques. To be fair, this view is also a bit optimistic for StdGP and DynDepth, since these techniques also reject the individuals that break the (static or dynamic) depth limit, although they do not create new individuals to replace them, and StdGP does not even evaluate the rejected individuals.

#### 6.2.2 Pessimistic View

Figure 6.2 shows an alternative way of representing the evolution of the best fitness during the run, for all the problems. Fitness is not plotted against the generations,
but against the number of individuals visited so far, where “visited” means created, evaluated, or simply reused. For each rejected individual, another one must be visited.

Looking at Figs. 6.1 and 6.2 we are tempted to conclude that the good results of DynOpEq only come at the expense of unacceptably high computational effort (except for the Artificial Ant problem). However, this view of the results is too pessimistic for DynOpEq, because it does not take into account the length of the rejected individuals. In Section 8.3 we will take a deeper look at the distribution of the rejected individuals and the way it relates to the self-adapting targets. This analysis will reveal that most of the DynOpEq rejections happen in the beginning of the run and refer to very small individuals, unveiling a much brighter scenario where simple modifications of DynOpEq can dramatically improve its efficiency. Ironically, this pessimistic view is, at the same time, too optimistic for MutOpEq, as it does not take into account the time spent on the successive mutations needed to adapt the individuals to the target.

6.2.3 Realistic View

Going back to the subject of what defines a good result, we should stress that fitness is not the only important element. Most GP users also want simple solutions, as they are easier to understand. So the goal is to find short programs that provide accurate results. This is, after all, what bloat is all about: the relationship between size and fitness.
Figure 6.2: Best fitness versus number of individuals visited, for all benchmark problems

Figure 6.3 shows, for all problems considered, the best fitness plotted against the program length. These are somewhat unconventional plots, but they provide a clear view of how varied the bloat dynamics is among the different problems and the different techniques. There is an implicit downwards timeline along the fitness axis. Depending on how fast the fitness improves with the increase of program length, the lines in the plot may point downward (south), or they may point to the right (east). Lines pointing south represent a rapidly improving fitness with little or no code growth. Lines pointing east represent a slowly improving fitness with strong code growth. Lines pointing southwest (bottom left) represent improvements in fitness along with a reduction of program length. We want our lines to point as south (and west) as possible.

As shown in Fig. 6.3, for all except the Regression problem DynOpEq exhibits a more downwards tendency than the other techniques. Even if this tendency is not apparent in the earlier stages of the run, it quickly becomes very marked and decisively leads the search towards short and accurate solutions. The results of MutOpEq are inconclusive. On the one hand, it seems like MutOpEq behaves much more similarly to all the other techniques than to DynOpEq; on the other hand, we suspect MutOpEq must be allowed to run for more generations before it reveals its distinctive behavior (and we can probably say the same about DynDepth and Double). One thing is obvious, though. In problems like the Artificial Ant and Parity, and in particular Multiplexer,
most of the non-equalising techniques finish their runs exhibiting such a behavior that it seems impossible they will ever reach the same fitness/length ratio as DynOpEq, no matter how many more generations they are allowed to run.

The Regression problem is the exception where DynOpEq used significantly longer programs than all the other techniques (except NoLimits). In terms of fitness there were no significant differences. It is common in symbolic regression problems for programs to get larger with very small fitness improvements, bloat being caused mostly by unoptimized, not inviable (see Section 2.2), code. However, the increased program length observed with DynOpEq seems to be of a different nature, as explained in 8.3.
Chapter 7

Real World Results

Here we present a summary of the experiments performed and the results obtained by OpEq in different real world applications. We focus on two problems of drug discovery and one problem of classification of satellite images, all regression problems of different known difficulty degrees. This provides an interesting test bench for OpEq, since the Regression benchmark was clearly the exception to the quality of the DynOpEq results (Section 6.2). Once again all the claims of significant differences in the results are supported by statistical evidence as described in Section 6.1 (Tools).

7.1 Drug Discovery

7.1.1 Experiments

Problems. We tested the OpEq techniques on two drug discovery problems. Based on the molecular structure of a set of candidate drug compounds, the first problem was to predict the human oral bioavailability of the compounds, and the second was to predict their toxicity. Both are addressed as symbolic regression problems, using a function set containing only the four main binary arithmetic operators and measuring fitness as the root mean squared error between outputs and targets. From previous work these were considered to be very hard problems where bloat and overfitting are major issues [10, 11]. For details see also [114] and [130].

Techniques. Both DynOpEq and MutOpEq were tested using only bin width 1, and compared with StdGP. For details see Section 6.1 (Techniques). From each problem data set, 30 random partitions were created with 70% of the samples for training and 30% for testing.

Parameters. A total of 30 runs were performed with each technique for each problem, each run using a different data partition. All the runs used populations of 500 individuals allowed to evolve for 100 generations. Tree initialization was performed with the Ramped Half-and-Half method as in Section 6.1 (Parameters). Selection for
reproduction used Lexicographic Parsimony Pressure [71] tournaments of size 10. The reproduction rate was 0.1. Mutation and crossover were used with probabilities 0.1 and 0.9, respectively. Selection for survival was not elitist. For more details consult the settings in [114] and [130].

**Tools.** The experiments were performed with GPLAB and statistical significance was determined with non-parametric ANOVAs. For details see Section 6.1 (Tools).

### 7.1.2 Results for Bioavailability

Figure 7.1 shows the training and test fitness plotted against the cumulative number of nodes used by all the individuals of the population during the run, for the bioavailability problem. It is yet another way of presenting the results, different from either optimistic or pessimistic view of Section 6.2. It is still optimistic for DynOpEq since it ignores the rejected individuals, but more informative than just plotting fitness against generations. As can be seen, DynOpEq exhibits the best learning curve (a) but also the worst generalization curve (b), revealing a certain degree of overfitting. MutOpEq reveals a somewhat similar behavior to StdGP in terms of learning speed and generalization ability. However, Fig. 7.2 clearly shows that MutOpEq is considerably better than StdGP in terms of the relationship between program length and fitness, i.e., in terms of bloat (these are the unconventional plots described in Section 6.2.3). While StdGP increases program length along the entire run, both OpEq techniques reach a point (more precisely, one distinct point for each technique) where program length remains stable while the training fitness continues to improve (a). It is precisely at this point that the test fitness of DynOpEq reaches the best value before it starts worsening (b).
7.1.3 Results for Toxicity

Figure 7.3 shows the training and test fitness plotted against the cumulative number of nodes used by all the individuals of the population during the run, for the toxicity problem. DynOpEq is once again the fastest learner (a) but also the fastest to overfit (b). MutOpEq resists overfitting better, but neither of the techniques retains its generalization ability. Figure 7.4 reveals that, also in this problem, both OpEq techniques are able to stabilize program length while continuing to improve the training fitness (a). Unlike in the previous problem, for DynOpEq the stabilization point is not where the test fitness reaches its best value (b).
7.2 Classification of Satellite Images

7.2.1 Experiments

Problems. We tested the OpEq techniques on a problem of classification of satellite images. Based on multispectral data, the goal was to identify burned areas in a forested/wooded savanna tropical ecosystem. We addressed this task also as a regression problem, using the four main binary arithmetic operators. Fitness is measured as the root mean squared error between outputs and targets, and a cutoff is applied to each output to obtain the classification. This problem is easily solved by simple classification trees (CART). For details see [115].

Techniques. DynOpEq and MutOpEq compared to StdGP (and all compared to CART). For details see Section 6.1 (Techniques). From the data set of this problem, 30 random partitions were created with 70% of the samples for training and 30% for testing.

Parameters. A total of 30 runs were performed with each technique, each run using a different data partition. All the runs used populations of 500 individuals allowed to evolve for 200 generations. The remaining parameters were the same as for the drug discovery problems (Section 6.1), adopted from [114] and [130], except that now the selection for survival is elitist, in the sense that the best individual of each generation is granted survival to the next generation.

Tools. The experiments were performed with GPLAB and statistical significance was determined with non-parametric ANOVAs. For details see Section 6.1 (Tools).

7.2.2 Results

Because there was no overfitting, we do not plot the training fitness for this problem. Instead, Fig. 7.5 shows the evolution of the average program length along the generations (a) and the relationship between average program length and test fitness (b). Once
Figure 7.5: Average program length versus generations (a) and test fitness versus average program length (b)

Figure 7.6: Boxplot of the accuracy values on the original and noisy data sets. MutOpEq has additional outliers at 95.2% (original) and 92.9% (noisy), not shown again, both OpEq techniques reach a stabilization point that, unlike before (see Sections 7.1.2 and 7.1.3), appears to be the same for both techniques. StdGP once again increases program length along the entire evolution. Figure 7.6 shows the accuracy achieved by each GP technique (and CART) on the original data set and on a modified noisy data set. StdGP was the only technique whose accuracy was not significantly affected by the noise.
Chapter 8

Discussion

We now provide a deeper analysis of the results presented in the previous two chapters. It is not our goal to detail the particular achievements of each technique, but instead to contrast the behavior of OpEq with that from other bloat control techniques, and analyze what makes them better or worse than the others. We begin by highlighting some differences between the two OpEq variants. Then we alert to the surprising realization that the effective bloat control provided by both DynOpEq and MutOpEq does not improve, and even seems to cause a negative side effect, on the generalization ability of the proposed solutions. Finally we provide a deeper analysis of the inner dynamics of DynOpEq suggesting causes and solutions for some of its issues.

8.1 MutOpEq versus DynOpEq

The results of the previous chapters reveal that OpEq is different from other bloat control techniques. Its main distinctive skill is the ability to stop code growth while continuing to improve fitness. This is the exact opposite of bloat, and OpEq achieves it without reverting to special genetic operators of program length limits. DynOpEq clearly demonstrates this ability in all benchmark problems except Regression (see Fig. 6.3, page 27, and see Section 8.3 for a possible explanation), but also in all real world problems, which happen to be regression problems (see Figs. 7.2, 7.4 and 7.5, pages 30, 31 and 32, respectively). MutOpEq is less successful in the benchmarks, but equally good in the real world problems.

MutOpEq was the first attempt to improve the efficiency of DynOpEq and, as such, it is the most recent and less studied variant. Although it appears to be much more computationally efficient than DynOpEq, there may be a number of disadvantages in its implementation. To begin with, performing several mutations in a single individual is also computationally expensive, although not as much as repeatedly creating, evaluating, and then rejecting so many individuals as DynOpEq does. The fact that in MutOpEq most individuals are not evaluated before being mutated is, however, the main cause of concern when comparing with DynOpEq. In DynOpEq good individuals do not go unnoticed, and are used to quickly increase the search bias towards the better
lengths without even having to wait for the next generation. In MutOpEq there is no such dynamism, as the target is seldom overridden. Because most created individuals are not evaluated, we may even be in the presence of an optimal individual but, just because it falls on an existing bin that happens to be full, we spoil its perfect genotype by adding or removing branches. The simple fact that many individuals are “mutilated” like this, although we try to minimize the impact that these mutations have on fitness, may well disturb the evolutionary dynamics of a normal GP run, and also what could be the typical behavior of OpEq. Hence MutOpEq does not completely resemble one or the other.

However, MutOpEq was very successful in all the regression problems where it was tested. Although a slow learner, it shows good prospects of achieving the same fitness using even shorter individuals than DynOpEq, if given enough generations. It may simply be that in regression problems the disruption caused by the mutations is not so large as in other problems like, for example, the Artificial Ant. If that is the case, it is possible that MutOpEq can be greatly improved by adopting more careful rules on how and where to mutate the individuals (see Section 9.1) so that the impact on their fitness is truly minimal. The success of OpEq is not just a question of having the right length distribution, but a question of having the right individuals (in terms of fitness) in the right place (in terms of length distribution).

8.2 Bloat and Overfitting

It has been generally accepted that code growth and overfitting are strongly related phenomena. In fact, a common agreement of many researchers is the so-called minimum description length principle (see for instance [98]), which states that the best model is the one that minimizes the amount of information needed to encode it. Simpler solutions are thought to be more robust and generalize better [101], while complex solutions are more likely to incorporate specific information from the training set, thus overfitting it. However, as mentioned in [30], this argument should be taken with care as too much emphasis on minimizing complexity can prevent the discovery of more complex yet more accurate solutions.

With OpEq the search is completely driven by fitness. As long as the result is better fitness, the complexity is allowed to increase freely, unrestrained by any maximum limits. So the balance between fitness and complexity is not controlled by any parameters, but dictated by the particularities of the fitness landscape and the evolutionary process itself. In principle, this should be enough to avoid bloat, as much as the fuzzy definition of bloat allows it. But is it enough to prevent overfitting? Obviously not. The search process will continue to look for better fitness, and in doing so it will be allowed to continue increasing the complexity of the solutions. Unbounded by any maximum limits, it is expected that overfitting will occur, sooner or later, as the solutions become more and more specialized for the training data. So, there can be overfitting without bloat. In fact, this seems more likely than having overfitting when there is bloat! After all, bloat stagnates the evolution, preventing the search process from finding better solutions. This means there is no further specialization, and no reduction of the generalization ability. So there can be bloat without overfitting, and overfitting without
bloat, and the most unlikely scenario seems to be the occurrence of both. Although arguable, these considerations are partially supported by our results. In the first drug discovery problem (Section 7.1.2) we can see that DynOpEq is the only technique that overfits (Fig. 7.1b). StdGP, on the other hand, is the only technique that bloats (Fig. 7.2a), at least when compared to the others. Summing up, there is one technique that bloats without overfitting (StdGP) and one technique that overfits without bloating (DynOpEq). None of the techniques does both. MutOpEq seems to fulfill the remaining combination (no bloating, no overfitting), but the search performed by this technique appears to have been stopped too soon. Given enough running time, it could continue improving the training fitness, specializing and overfitting just like DynOpEq, or unlikely it could stagnate in terms of fitness and program length, never bloating or overfitting, but never finding any better solutions either. In the second drug discovery problem (Section 7.1.3) we see that all techniques overfit (Fig. 7.3b) and we can once again say that StdGP bloats (Fig. 7.4a). This may be used as evidence against our considerations, arguing that overfitting is not related to anything else except the learning itself, i.e., the faster a technique learns, the faster it overfits. However, the results obtained in the satellite image classification problem (Section 7.2.2) once again raise a suspicion regarding the role of redundant code in overfitting. Although none of the techniques overfitted and only StdGP bloated (as revealed by Fig. 7.5b) StdGP was the only one whose accuracy was not significantly affected by the noisy data.

8.3 Dynamics of DynOpEq

From all the results presented so far, DynOpEq seems to be the most successful OpEq variant, or at least the one that consistently achieves better results in different types of problems. There is the exception of the Regression benchmark, and there is the major efficiency problem caused by the high number of rejections. We now reveal some of the inner dynamics of DynOpEq and identify possible causes and solutions for both issues.

8.3.1 Rejections and Target Distributions

In Figs. 8.1, 8.2 and 8.3 the left plot (a) refers to DynOpEq with bin width 1 in the Regression problem, and the right plot (b) refers to DynOpEq with bin width 10 in the Parity problem. The Regression problem was chosen for the exceptionally bad results provided by DynOpEq, and the Parity problem was chosen for comparative purposes. Figure 8.1 shows the number of rejected individuals by length class and generation. Figure 8.2 shows the desired number of individuals by length class and generation, i.e., the evolution of the target distribution of program lengths during the run. Figure 8.3 shows the evolution of the actual distribution of program lengths during the run. Each value is the mean over the 30 runs. There is considerable variance between the different Regression runs, but not much in the Parity case.

Beginning with the Regression problem, one interesting fact is immediately apparent on Fig. 8.1a. The “number of rejections” axis does not go beyond the value 1000 for visualization purposes, but the number of rejected individuals in the conspicuous peak
Figure 8.1: Rejections. Regression with bin width 1 (a) and Parity with bin width 10 (b)

Figure 8.2: Target distributions. Regression with bin width 1 (a) and Parity with bin width 10 (b)

Figure 8.3: Actual distributions. Regression with bin width 1 (a) and Parity with bin width 10 (b)
Table 8.1: Percentage of rejections falling outside the target distribution

<table>
<thead>
<tr>
<th>Bin width</th>
<th>Regression</th>
<th>Artificial Ant</th>
<th>Parity</th>
<th>Multiplexer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.6%</td>
<td>5.0%</td>
<td>13.1%</td>
<td>3.8%</td>
</tr>
<tr>
<td>5</td>
<td>0.5%</td>
<td>4.1%</td>
<td>4.9%</td>
<td>2.8%</td>
</tr>
<tr>
<td>10</td>
<td>0.003%</td>
<td>4.6%</td>
<td>4.0%</td>
<td>1.8%</td>
</tr>
</tbody>
</table>

reaches higher than 10 thousand. If completely drawn, this peak would be roughly 10 times higher than shown. This reveals that in the Regression problem there is a concentration of rejections of small individuals in the beginning of the run. More specifically, with a bin width of 1, roughly 16% of the total number of rejections happen in the first 5 generations and refer to individuals with sizes between 1 and 10. With bin widths of 5 and 10 this percentage rises dramatically to 75–76%. If drawn, these peaks would be approximately 350 times higher than shown! In the Multiplexer problem (not shown) these percentages are also relatively high (10%, 23%, and 40% for bin widths of 1, 5, and 10). This is a good indication that the high number of visited individuals represented in the pessimistic view of Section 6.2.2 may not be so dramatic. After all, such small individuals are not very computationally expensive. However, for the Artificial Ant and Parity problems the initial peaks only contain 0.7–1.9% of the rejections. Figure 8.1b shows the modest initial rejection peaks for the Parity problem with bin width of 10.

Apart from the initial peaks, there is a concentration of rejections that seems to follow a path along the generations (Fig. 8.1, both plots). As we can see in Fig. 8.2, for both problems this roughly falls on the path formed by the moving target distribution of program lengths. The same happens for the remaining problems (not shown). One could suspect that, since most rejections happen on bins that already exist, many individuals are accepted into newly created bins outside the boundaries of the target. This is, however, incorrect. When comparing the actual length distributions on Fig. 8.3 with the desired targets on Fig. 8.2 we see they are close matches, except for the undesired peak of the actual length distribution in the Regression problem (Fig. 8.3a, peak reaches the value 130). The same happens for all problems with all bin widths: a close match between actual and desired distributions, except for some traces of the highest initial rejection peaks.

The knowledge that most rejections happen in existing bins within the target, along with the fact that the target is, for the most part, not overridden, provides the perfect setting for improving the efficiency of the DynOpEq technique (see Section 9.1). We have measured the percentage of rejections that actually fall outside the boundaries of the target. The numbers are presented in Table 8.1.
8.3.2 Rejections and Function Sets

An additional observation can be made regarding the rejection plot of the Regression problem (Fig. 8.1a), which is the jagged look of the rejection pattern. This characteristic is almost absent, but still detectable, in the target length distribution (Fig. 8.2a). The Parity problem with bin width 1 exhibits an even stronger jagged surface (see Fig. 8.4a), with absolutely no rejections for the even length bins. However, the target for this problem is also completely jagged (see Fig. 8.4b). This makes total sense, as the function set of the Parity problem does not allow the creation of even sized individuals, so the mean fitness of those bins is actually null, and no individuals are expected there. In the Regression problem, despite the mixed arity function set used, there seems to be some difficulty in creating individuals of even size. But because they do appear in the population, the mean fitness of those bins is not null, and this is reflected in the relatively smooth surface of the target distribution. Even more interesting, as we can see in Fig. 8.3a the actual distribution of program lengths observed during the run is equally smooth, meaning that DynOpEq is actually inserting these difficult individuals in the population. In StdGP both the (hypothetical) target and actual distributions of program lengths are jagged (see Fig. 8.5). In both cases many more odd sized individuals are created than even sized ones, but with DynOpEq the most requested bins fill up sooner and start increasing their number of rejections, while the least requested bins still have room available. This is what causes the jagged pattern of rejections. In StdGP the most common sizes fill the population leaving no room for the difficult ones. By granting the rare individuals a place in the population, DynOpEq is actually increasing the genotypic diversity, that indeed reaches the highest levels of all the tested techniques (see Fig. 8.6a). On the other hand, this increased diversity is accompanied by the proliferation of inviable code (see Fig. 8.6b), as if introns were being used to fill the nodes needed to build even sized individuals. This could explain why the jagged pattern is more pronounced for the small sizes, where there is not enough room for introns. It could also explain the atypical proliferation of inviable code in the Regression benchmark, and ultimately the poor results achieved by DynOpEq in this problem. We hypothesize that this issue does not apply to the real world regression problems because their function sets only contain binary operators.
Figure 8.4: Rejections (a) and target distribution (b) for Parity with bin width 1

Figure 8.5: Hypothetical target (a) and actual (b) distributions for Regression with bin width 1, using StdGP

Figure 8.6: Diversity (a) and percentage of inviable code (b) for Regression
Chapter 9

The Future

This report has presented a historical view of the research done in the subject of bloat in Genetic Programming. After reviewing past theories and available control methods, a description of the latest bloat developments was presented. An introduction to the new theory, Crossover Bias, and a detailed explanation of the new control method, Operator Equalisation, have been followed by a summary of experiments that are currently being done in different benchmark and real world problems. The results promise to revolutionize the field, as they reveal novel evolutionary dynamics that allow successful search without code growth. The remaining paragraphs are dedicated to describing the future improvements and extensions of current work that will allow Operator Equalisation to become an integral part of regular GP systems.

9.1 Improvements

**DynOpEq.** The most obvious item in need of improvement is the (in)efficiency of DynOpEq, caused by the high number of individuals that are created, evaluated, and then simply rejected and never used again. The analysis presented in Section 8.3.1 has revealed that most rejections happen in existing bins within the target distribution. So there is a minor modification of DynOpEq that can provide a major breakthrough in its efficiency. In the pseudocode of Fig. 5.1 (page 22), if line 6 of the validation procedure (“if DynOpEq and i is the new best-of-bin”) is replaced by simply “if DynOpEq”, the need to evaluate all the individuals attempting to enter existing bins disappears. The bins will accept individuals while they have room available, and will reject them once full. Only the individuals who fall outside the boundaries of the target will have to be evaluated. And because Section 8.3.1 also revealed that the target is barely overridden anyway, this modification will hardly lead to significant changes in the actual length distribution. To understand the full impact that this simple modification can have on the efficiency of DynOpEq, see in Table 8.1 (page 37) the low percentage of rejections that would actually require evaluation of the individual. However, unlike the current implementation, this modification entails the danger of rejecting good individuals.
**MutOpEq.** As for MutOpEq, the mutations applied to the individuals when adapting them to fit a given bin may destroy successful individuals that maybe took several generations to find. Although the dynamics of the MutOpEq mutations have not yet been studied, it is expected that the more they change the fitness of the individuals, the larger is their disruptive effect on the evolutionary process. Given so, there are a few ideas to make these mutations as neutral as possible. The obvious one is to mutate only within inviable code, but inviable code is usually hard to find in regression problems, and the implementation of “smart” mutations that minimize fitness changes may be computationally expensive. Another approach is to delay the mutations as much as possible, saving all the individuals that need adaptation until there are as many saved individuals as the remaining positions in the target, and only then decide which individuals are closer to which bins. This would allow the minimization of the number of nodes mutated, and even permit an informed decision of whether to mutate certain individuals or leave them intact. This decision could even be based on fitness, since there would be a limited number of individuals to evaluate. The individuals not worth mutating could remain intact (if they’re good) and override the target, or could be replaced (if they’re bad) by other individuals, possibly some parents from the previous generation.

**Hybrid.** Joining ideas from both OpEq variants, a hybrid scenario immediately emerges. Each individual that does not fit the target can be either rejected, mutated, substituted, or accepted, depending on an array of possible factors. These factors may be related not only to length and fitness, but also to how much of the entire target has already be filled and how hard it is becoming to fill the remaining positions.

### 9.2 Extensions

Regardless of the implementation details, the main idea behind OpEq is to have the search follow a length distribution that follows fitness. This can be generalized and integrated into different stages of the evolutionary process. Indeed, many of the bloat control methods described in Chapter 3, either acting at the evaluation, selection, breeding, survival, or other level of the evolution, can readily benefit from this idea. What follows is a selection of methods from Chapter 3, each acting at a different level, with a brief explanation of how this can be done.

**Evaluation.** In Parametric Parsimony Pressure (Section 3.1, page 7) the fitness of each individual is a function of its raw fitness and its size. It is very hard to find the right setting of the parsimony coefficient and keep it correctly updated during the run, as it depends not only on the problem but also on the particular demands of each evolution phase. Using the OpEq idea, the fitness of the individual could be a function of its raw fitness and a static (and still undefined) measure of “how well the individual fits the target”. This function would not need the parsimony coefficient because the target already incorporates and constantly adapts to the needs of the evolutionary process.
Selection. In Double Tournament (Section 3.2, page 8) one of the tournaments is based on size, and uses a parameter to determine the probability of selecting the smaller or larger individual. Once again this parameter is hard to tune, for the same reasons stated for the parsimony coefficient of the Parametric Parsimony Pressure method. Based on OpEq, the tournament based on size could discard the probability parameter and once again rely on a measure of how well the individual fits the target.

Breeding. Size Fair Crossover (Section 3.3, page 9) is one of many variants of crossover that can integrate OpEq. After choosing the crossover point on the first parent, the second crossover point should be such that the offspring is guaranteed to fit the target.

Survival. In Resource-Limited GP (Section 3.4, page 11) there is a dynamic limit on the total amount of nodes that the entire population (not each individual) can use, resulting in automatic population resizing. Candidates to the new generation are queued by fitness and resources (the nodes they require) are allocated on a first come, first served basis, skipping the individuals for whom there are no more nodes available. Instead, OpEq could be used to skip the individuals who do not fit the target.

Others. The Waiting Room method (Section 3.5, page 11) also implements a queue, where individuals must wait before entering the population, with a waiting time directly proportional to their size. Instead, the waiting time could be inversely proportional to how well they fit the target.

Basically, the integration of OpEq into the different elements of GP is based on substituting the concept of size, or length, by the concept of target length distribution. The size of each individual is then rated, not according to its own value, but according to how it can contribute to what is expected from the population as a whole.

9.3 Final Considerations

Theoretically or empirically motivated, more or less successful, all the previous bloat control methods were a valid contribution to the research on bloat. Operator Equalisation, however, is not just another bloat control method. It is a new, population based, way of maintaining the search within the lengths that are known to be worth exploring. In principle, regular selection should be able to do this, but it does not. The success of Operator Equalisation validates the Crossover Bias theory and confirms that the problem is not with selection, but with selection having to deal with the length bias (and consequent fitness bias) caused by the genetic operators. Taking one step back to once again look at the original inspiration of Genetic Programming, one realizes that the natural biological populations avoid this problem entirely. Nature implements a very strict pre-selection of individuals before they ever reach adulthood. The weak individuals do not survive childhood, never reaching the age when they would start competing for the right to breed. In nature, the selection for reproduction only deals with individuals of
similar fitness. Operator Equalisation is providing the cue to go back and find out what
may have been missing from Genetic Programming all along. Some of the very first
bloat control methods, like the brood recombination by Tackett [126], may have passed
unnoticed just because they were being interpreted within old theories that turned out
to be wrong or incomplete. Maybe it is time to find out, in the context of the Crossover
Bias theory, how close they really were to the answer.

Acknowledgements

The authors acknowledge Sean Luke for suggesting the Minimum Child Size experi-
ments described in Chapter 4 and Riccardo Poli for the permission to use the Lagrange
distribution graphs presented in the same chapter. Thank you also to the anonymous
reviewers of [111] for providing some of the references of Chapter 3. The first and
third authors also acknowledge project “EnviGP – Improving Genetic Programming
for the Environment and Other Applications” (PTDC/EIA-CCO/103363/2008) from
Fundação para a Ciência e a Tecnologia, Portugal.
Bibliography


[10] F. Archetti, S. Lanzeni, E. Messina, L. Vanneschi, Genetic programming and other machine learning approaches to predict median oral lethal dose (LD50) and plasma protein binding levels (%PPB) of drugs. In Proceedings of EvoBIO-2007, ed. by E. Marchiori et al. (Springer 2007), pp. 11–23


