MSP and NCFP: Novel bloat control methods for Genetic Programming

Marko Šmid, Matej Moravec, Miha Ravber

University of Maribor, Faculty of Electrical Engineering and Computer Science,
Koroška cesta 46, 2000 Maribor, Slovenia
E-mails: {marko.smid2, matej.moravec, miha.ravber}@um.si

Abstract. *Bloat in Genetic Programming refers to the phenomenon where evolved solutions become excessively large without corresponding improvements in fitness, resulting in diminished solution quality and reduced readability. In this study, we propose two novel bloat control methods, called Minimal Subtree Pruning and Node Count Frequency Pruning. The first method constrains the solution size by pruning subtrees such that the resulting number of nodes is as close as possible to a predefined maximum. The second method, designed specifically for solutions encoded as Behavior Trees, reduces the bloat by removing nodes with count frequencies below a predefined threshold. Genetic programming algorithms utilizing these methods are evaluated against a standard genetic programming algorithm with a commonly used depth-limiting bloat control method. The Collector problem domain is utilized as a benchmark. The experimental results indicate that both proposed methods improve the overall performance of the genetic programming algorithm, producing better and more compact solutions compared to the baseline algorithm.*

1 Introduction

Genetic Programming (GP) is a branch of evolutionary algorithms in which solutions are represented as computer programs. Unlike fixed-length representations common in other evolutionary techniques, GP solutions can assume arbitrary structural forms. In standard GP, solutions are typically encoded as tree structures composed of functions and terminals [1].

GP employs a genetic algorithm that simulates the process of natural evolution, where a population of candidate solutions is refined iteratively over a predefined number of generations. During each generation, genetic operators are applied, such as selection, crossover, recombination, and mutation. The selection process identifies high-performing solutions as candidates for the next generation. The selected solutions may undergo modifications through crossover, which combines parts of two solutions, or mutation, which introduces random changes to nodes or subtrees. Other solutions are retained unchanged through reproduction. Each solution is then evaluated, to assess its performance on a given benchmark. This evolutionary process continues until a specified stopping criterion is met.

The variation operators in GP often modify solutions by exchanging or inserting structural components within programs. Due to the nature of these modifications, the evolutionary dynamics tend to favor the accumulation of additional code rather than its removal. This is because the introduction of new material is more likely to maintain or improve fitness slightly than the elimination of existing components, which may risk degrading performance. As a result, solutions often grow in size and depth progressively over future generations, even when this growth no longer yields meaningful improvements in fitness. This tendency for solution structures to increase in complexity without corresponding gains in performance is known as code bloat [2].

Bloat is a well-known problem, yet no consensus exists on why it occurs. Nevertheless, several theories have been proposed to explain its emergence. Early theories focused on the role of introns. An intron is a part of the solution that does not affect its performance. One of the earliest explanations, known as the *hitchhiking* theory, suggests that introns are preserved because they are linked to essential building blocks of the solution, and since they do not impact fitness negatively, there is no evolutionary pressure to remove them [3]. This idea led to a broader theory, called the *defense against crossover* theory, which presumes that introns serve as a protective buffer, shielding the functional code from the potentially disruptive effects of variation operators once a high-performing solution is discovered [4]. The *removal bias* theory highlights an asymmetry in how genetic operations affect code regions. Specifically, to exploit crossover within non-functional (intron) code regions without degrading fitness, the size of the removed subtree must be constrained to the intron region [4]. In contrast, the *fitness causes bloat* theory takes a non-intron-based perspective, arguing that larger solutions are statistically more likely to exhibit higher fitness [2].

Although no formal method for bloat control exists, a wide range of techniques have been proposed to mitigate its effects in GP [5]. These approaches can be categorized broadly into three main groups. The first group includes size and depth limiting techniques, which impose explicit constraints on the maximum size or depth of solutions to prevent the evolution of excessively large solutions [1]. The second group encompasses penaliza-

tion methods, where overly large solutions are assigned reduced fitness scores, thereby discouraging their selection during evolution [6, 7]. The third group involves the use of modified genetic operators that are designed explicitly to reduce or manage growth during the variation process [8, 9]. In addition to these techniques, various alternative methods have been proposed to control bloat. These include The waiting room [10], Death by size [10], Prune and Plant [11], and Pareto-based Multi-objective Parsimony Pressure [12].

There are several reasons for employing bloat control methods in GP. First, controlling bloat can lead to more compact and interpretable final solutions without compromising and potentially even improving performance. Second, by limiting the size of solutions, the effective search space may be reduced, enabling the algorithm to discover high-quality solutions with smaller representations. Third, smaller solutions typically require fewer computational resources, which can result in faster execution and reduced processing time during evolution [13].

This study proposes two novel bloat control methods called Minimal Subtree Pruning (MSP) and Node Count Frequency Pruning (NCFP). MSP constrains solution size by pruning subtrees such that the resulting number of nodes is as close as possible to a predefined maximum. NCFP reduces bloat by removing nodes with count frequencies below a predefined threshold. We evaluated the proposed methods against a commonly used depth-limiting method [1], which served as a baseline. Throughout the paper, this method is referred to as Tree Depth Pruning (TDP). TDP prunes solutions that exceed the maximum depth after crossover and mutation.

The remainder of this paper is structured as follows. The following section provides a detailed description of the proposed methods. Section 3 outlines the experimental setup employed. Section 4 presents and discusses the experimental results. Finally, Section 5 summarizes the conclusions and identifies directions for future work.

2 Bloat control methods

This section describes our proposed methods, MSP and NCFP. The main idea of the MSP is to prune the solution by removing a subtree, where the size of the solution after the subtree is removed is closest to the maximum defined tree size, determined by the number of nodes. This method follows the rule that removing the fewest nodes possible will result in the least performance changes. The maximum number of nodes that a solution can have is defined by the *TreeSize* parameter. If a solution exceeds the *TreeSize*, a suitable subtree must be removed. For each possible subtree, the difference is calculated between the modified solution size and the maximum tree size. In the next step, a subtree with the smallest difference is removed, where the number of nodes is below or equal to the maximum number of nodes. When two or more candidate subtrees have an identical number of nodes, the first subtree encountered is selected in a left-to-right traversal. If the root of the subtree was the only child of a function node, then this node is replaced by a random terminal. This method is applied before the evaluation process. One drawback of this method is that no information about the removed subtree is known, and its impact on the overall performance of the solution.

The NCFP method is tailored specifically for Behavior Trees (BTs) [14], and operates on the premise that nodes with low execution frequencies contribute minimally to the overall performance of a solution and can thus be removed with greater confidence. This method is applied post-evaluation. The frequency of each node's execution (termed node count frequency) is recorded during the evaluation phase. Nodes with count frequencies below a specified threshold, defined by the parameter *pruneThreshold*, are identified as candidates for removal. For each candidate node, a random number is sampled uniformly from the interval $[0, 1)$. The node is removed if the random number is less than the *pruneProbability*. These two parameters, *pruneThreshold* and *pruneProbability*, allow a high degree of flexibility and control of the pruning process. A notable drawback of NCFP is the requirement for reevaluation of solutions when nodes are removed with non-zero count frequencies.

3 Experiment design

To evaluate the performance of the proposed MSP and NCFP methods, we conducted experiments within the Collector problem domain. The Collector game requires an agent to navigate a dynamic terrain and collect as many targets as possible within a specified time limit. The time limit was set to 60 seconds, and the best fitness that an agent can achieve is -550. The experiments were conducted using the General Intelligent AgeNt Trainer (GIANT) platform [15]. GIANT's modular design allowed for seamless integration of our custom bloat control methods. The platform's support for parallel processing and highly configurable parameters facilitates large-scale experimentation, reducing computational time substantially while maintaining reproducible results.

A standard GP algorithm was employed, augmented with an elitism mechanism. Elitism guarantees that the best solutions are retained in the population, thereby maintaining solution quality throughout the evolutionary process [16]. BTs were used as a solution representation structure. The specific GP parameters utilized, including population size, crossover and mutation probability, and other relevant settings, are summarized in Table 1.

For each bloat control method, we tested several configurations, each described in Table 2.

4 Results and discussion

In the experiments, we compared the performance of the GP algorithm utilizing MSP and NCFP methods against a GP algorithm with a TDP bloat control method. Four different matrices were analyzed when comparing these methods, namely, mean best fitness, mean population fitness, mean population tree depth, and mean population tree size. Mean population fitness was used to assess the stability and generalizability of GP algorithms with bloat control methods. Unlike mean best fitness, which reflects

Table 1: GP algorithm parameter values.

Parameter	Value
Population size	100
Generations	20
Selection	tournament (size = 2)
Crossover prob.	0.75
Mutation prob.	0.25
Elitism prob.	0.05
Functions	Sequencer, Selector, Inverter
Terminals	RayHitObject, MoveForward, Rotate
Runs	30
Init. pop. gen.	random

Table 2: Parameters of the bloat control methods.

Method	Parameter values
TDP_25	max tree depth: 25
TDP_10	max tree depth: 10
TDP_6	max tree depth: 6
MSP_100	max tree depth: 25, max tree size: 100
MSP_50	max tree depth: 25, max tree size: 50
NCFP_20	max tree depth: 25, prune threshold: 20% prune probability: 0.2
NCFP_10	max tree depth: 25, prune threshold: 10%, prune probability: 0.2
NCFP_5	max tree depth: 25, prune threshold: 5%, prune probability: 0.4

only the top individual, it may reveal issues like stagnation or overfitting. All the values were averaged over 30 independent runs.

Figure 1 presents the results for the mean best fitness. The results show that the methods can be divided into three groups based on their final mean best fitness. The first group, achieving the best performance, consisted of NCFP_20, NCFP_5, and NCFP_10. The second group, demonstrating intermediate performance, included TDP_6 and MSP_50. The third group, exhibiting the lowest performance, comprised the TDP_25, TDP_10, and MSP_100. Among all algorithms evaluated, NCFP_10 yielded the best solutions consistently throughout most of the optimization process. However, in the final four generations, NCFP_5 surpassed NCFP_10, achieving better fitness. Notably, all the MSP and NCFP methods, except for MSP_100, outperformed the baseline method, underscoring their performance in controlling bloat while maintaining or improving solution quality.

The mean population fitness is represented in Figure 2. The results revealed a distinct partitioning of the evaluated methods into two groups based on the mean population fitness, contrasting with the trends observed in mean best fitness. The first group, demonstrating better mean population fitness, was comprised of TDP_10, TDP_6, MSP_100, MSP_50, and TDP_25. The second group, exhibiting lower mean population fitness, included NCFP_20, NCFP_5, and NCFP_10. Notably, the baseline TDP_25 outperformed all the other methods in this metric, highlighting its effectiveness in maintaining population-wide fitness despite the absence of bloat control

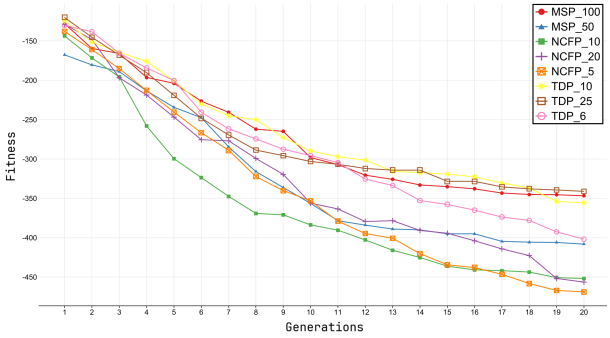

Figure 1: Mean best fitness per generation, averaged over 30 independent runs.

mechanisms.

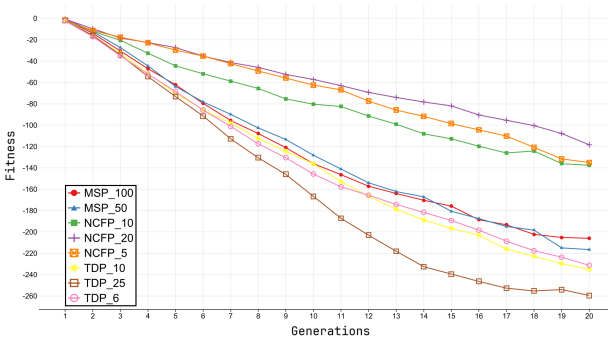

Figure 2: Mean population fitness per generation, averaged over 30 independent runs.

The third metric evaluated in this study was the mean population tree depth, with the results presented in Figure 3. As anticipated, TDP_6 generated solutions with the smallest tree depths consistently among the evaluated algorithms, consistent with its stricter depth constraints. Conversely, TDP_25 produced solutions with the largest tree depths, reflecting its more permissive depth limitations. The remaining algorithms, TDP_10, MSP, and NCFP variants, generated solutions with intermediate tree depths, ranging from 8 to 11, indicating a balanced approach to bloat control.

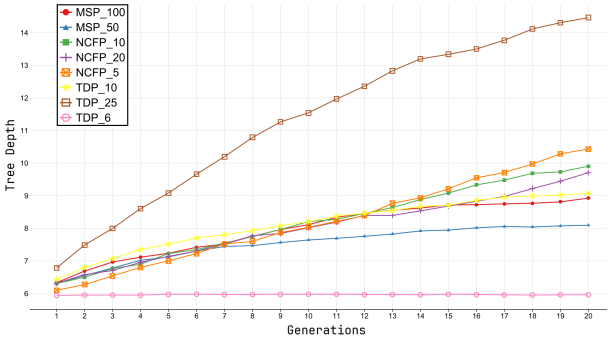

Figure 3: Mean population tree depth per generation, averaged over 30 independent runs.

The fourth and final metric assessed was the mean pop-

ulation tree size, with the results presented in Figure 4. In alignment with the mean population tree depth results, the TDP_25 generated solutions with the largest tree sizes. The remaining algorithms can be classified into three distinct groups based on tree size. The first group, exhibiting the smallest solution sizes, comprised MSP_50, NCFP_10, NCFP_20, and NCFP_5. The second group, with larger solution sizes, included TDP_6 and MSP_100. The third group, characterized by the second-largest solution size, consisted solely of TDP_10. A notable observation is that, although NCFP_5 generated solutions with tree depths exceeding those of most algorithms (except TDP_25), it yielded the smallest tree sizes consistently among all evaluated algorithms. This suggests that NCFP_5 balances depth and size effectively, achieving compact solutions without compromising structural complexity.

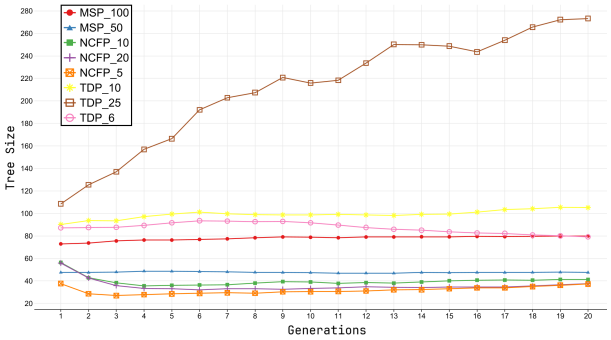

Figure 4: Mean population tree size per generation, averaged over 30 independent runs.

5 Conclusion

This study introduced two novel bloat control methods for GP: MSP and NCFP. These methods were designed to mitigate the excessive growth of solutions, enhancing both their performance and readability. Evaluated within the Collector problem domain using the GIANT platform, GP algorithms utilizing MSP and NCFP methods were compared against a GP algorithm with a depth-limiting bloat control method. The experimental results demonstrated that GP algorithms using the proposed methods, particularly the one with the NCFP method, outperformed the GP with the baseline method in terms of mean best fitness while producing more compact solutions, as evidenced by the reduced tree sizes and balanced tree depths. However, the baseline method excelled in mean population fitness, highlighting trade-offs in population-wide performance.

While MSP and NCFP showed strong performance, further comparison against other bloat control methods across diverse problem domains is required. In addition, systematic experiments under varying GP algorithm configurations (e.g., population sizes and time budgets) are necessary to evaluate their robustness and adaptability.

Acknowledgments

The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0114).

References

[1] J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA, USA: MIT Press, 1992.

[2] W. B. Langdon and R. Poli, “Fitness causes bloat,” in Soft Computing in Engineering Design and Manufacturing, P. K. Chawdhry, R. Roy, and R. K. Pant, Eds. London, UK: Springer-Verlag, 1997, pp. 13–22.

[3] W. A. Tackett, “Recombination, selection, and the genetic construction of computer programs,” Ph.D. dissertation, Univ. Southern California, Los Angeles, CA, USA, 1994.

[4] T. Soule and J. A. Foster, ”Removal bias: a new cause of code growth in tree based evolutionary programming,” 1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360), Anchorage, AK, USA, 1998, pp. 781-786, doi: 10.1109/ICEC.1998.700151.

[5] N. Javed, F. Gobet, and P. Lane, ‘Simplification of genetic programs: a literature survey’, Data Min Knowl Disc, vol. 36, no. 4, pp. 1279–1300, Jul. 2022, doi: 10.1007/s10618-022-00830-7.

[6] T. Soule and J. A. Foster, ”Effects of code growth and parsimony pressure on populations in genetic programming,” Evol. Comput., vol. 6, no. 4, pp. 293–309, 1998, doi: 10.1162/evco.1998.6.4.293.

[7] S. Luke and L. Panait, ”Lexicographic parsimony pressure,” in Proc. 4th Annu. Conf. Genet. Evol. Comput., 2002, pp. 829–836.

[8] W. B. Langdon et al., ”Size fair and homologous tree genetic programming crossovers,” Genetic Programming and Evolvable Machines, vol. 1, no. 1/2, pp. 95–119, 2000.

[9] H. Bhardwaj and P. Dashore, ‘A novel genetic programming approach to control bloat using crossover and mutation with intelligence technique’, in 2015 International Conference on Computer, Communication and Control (IC4), Sep. 2015, pp. 1–6. doi: 10.1109/IC4.2015.7375619.

[10] L. Panait and S. Luke, ”Alternative bloat control methods,” in Proc. Genetic Evol. Comput. Conf., Berlin, Heidelberg, 2004, pp. 630–641.

[11] E. Alfaro-Cid, A. Esparcia-Alcázar, K. Sharman, and F. F. de Vega, ‘Prune and Plant: A New Bloat Control Method for Genetic Programming’, in 2008 Eighth International Conference on Hybrid Intelligent Systems, Sep. 2008, pp. 31–35. doi: 10.1109/HIS.2008.127.

[12] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler, ‘Multiobjective genetic programming: reducing bloat using SPEA2’, in Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546), May 2001, pp. 536–543 vol. 1. doi: 10.1109/CEC.2001.934438.

[13] S. Luke and L. Panait, ”A Comparison of Bloat Control Methods for Genetic Programming,” in Evolutionary Computation, vol. 14, no. 3, pp. 309-344, Sept. 2006, doi: 10.1162/evco.2006.14.3.309.

[14] M. Mateas and A. Stern, ”A behavior language for story-based believable agents,” in IEEE Intelligent Systems, vol. 17, no. 4, pp. 39-47, July-Aug. 2002, doi: 10.1109/MIS.2002.1024751.

[15] GIANT, https://github.com/UM-LPM/GIANT

[16] D. Dumitrescu et al., Evolutionary Computation. Boca Raton, FL, USA: CRC Press, 2000.