Destructiveness of Lexicographic Parsimony Pressure and Alleviation by a Concatenation Crossover in Genetic Programming

Timo Kötzing[1], J. A. Gregor Lagodzinski[1], Johannes Lengler[2], and Anna Melnichenko[1]

[1] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
[2] ETH Zürich, Zürich, Switzerland

Abstract. For theoretical analyses there are two specifics distinguishing GP from many other areas of evolutionary computation. First, the variable size representations, in particular yielding a possible bloat (i.e. the growth of individuals with redundant parts). Second, the role and realization of crossover, which is particularly central in GP due to the tree-based representation. Whereas some theoretical work on GP has studied the effects of bloat, crossover had a surprisingly little share in this work.
We analyze a simple crossover operator in combination with local search, where a preference for small solutions minimizes bloat (*lexicographic parsimony pressure*); the resulting algorithm is denoted *Concatenation Crossover GP*. For this purpose three variants of the well-studied MAJORITY test function with large plateaus are considered. We show that the Concatenation Crossover GP can efficiently optimize these test functions, while local search cannot be efficient for all three variants independent of employing bloat control.

1 Introduction

Genetic Programming (GP) is a field of Evolutionary Computing (EC) where the evolved objects encode programs. Usually a tree-based representation of a program is iteratively improved by applying variation operators (mutation and crossover) and selection of suitable offspring according to their quality (fitness). Most other areas of EC deal with fixed-length representations, whereas the tree-based representation distinguishes GP. This representation of variable size leads to one of the main problems when applying GP: *bloat*, which describes an unnecessary growth of representations. Solutions may have many redundant parts, which could be removed without afflicting the quality, and search is slowed down, wasted on uninteresting areas of the search space.

In this paper we study GP from the point of view of run time analysis. While many previous theoretical works analyzed mutational GP with the offspring produced by varying a single parent, we analyze a GP algorithm employing a simple crossover with the offspring produced from two parents. Although our crossover is far from practical applications of GP (it merely concatenates the two parent trees), this simple setting aims at understanding the interplay between (our variant of) crossover, the problem of bloat and *lexicographic parsimony pressure*, a method for bloat control introduced in [14]. Other theoretical work in GP has analyzed different problems and phenomena, in particular for the Probably

Table 1: Overview of the results of the paper. A check mark denotes optimization in polynomial time with high probability, a cross denotes superpolynomial optimization time. A check mark with a subscript e denotes the results obtained experimentally.

	local search		**crossover**
problem class	**w/ bloat control**	**w/o bloat control**	**w/ bloat control**
$+c$-Majority	× Theorem 1	✓$_e$ Figure 2	✓ Theorem 10
2/3-Majority	✓ Theorem 2	✓$_e$ Figure 2	✓ Theorem 10
2/3-SuperMajority	✓ Theorem 5	× Theorem 6	✓ Theorem 13

Approximately Correct (PAC) learning framework [10], the Max-Problem [5,11,13] as well as Boolean functions [15, 16, 18].

For the effects of bloat in the sense of redundant parts in the tree, we draw on previous theoretical works that analyzed this phenomenon, especially [19] and [2]. In these, the fitness function Majority as introduced in [6] was analyzed. Individuals for Majority are binary trees, where each inner node is labeled J (short for *join*, but without any associated semantics) and leaves are labeled with variable symbols; we call such trees *GP-trees*. The set of variable symbols is $\{x_1, \ldots, x_n\} \cup \{\overline{x}_1, \ldots, \overline{x}_n\}$, for some n. In particular, variable symbols are paired: x_i is paired with $\overline{x}_i$. For Majority, we call a variable symbol x_i *expressed* if there is a leaf labeled x_i and there are at least as many leaves labeled x_i as there are leaves labeled $\overline{x}_i$; the positive instances are in the majority. The fitness of a GP-tree is the number of its expressed variable symbols x_i. This setting captures two important aspects of GP: variable length representations and that any given functionality can be achieved by many different representations. However, the tree-structure, typically crucial in GP problems, is completely unimportant for the Majority function.

We know that Majority can be efficiently optimized by a mutational GP called (1+1) GP (see Algorithm 1 for details, basically performing a randomized local search). This holds in the case preferring shorter representations by lexicographic parsimony pressure, as shown in [19], as well as in the case without such preference [2]. Similar to recent literature on theory of GP, we will consider lexicographic parsimony pressure as our method of bloat control and henceforth only speak of *bloat control* to denote this method. We note, however, that the GP literature knows many more methods for controlling bloat which is beyond the scope of our theoretical analysis.

In addition to weighted versions of Majority, another, similar fitness function Order (see also [3,20]) has been considered, but neither of these provide us with a strong differences in the optimization behavior of different GP algorithms. Thus, we propose three variants of Majority, called $+c$-Majority, 2/3-Majority and 2/3-SuperMajority, which negatively affect the optimization of certain GP algorithms.

For $+c$-Majority a variable is expressed if its positive literals are not only in the majority, but also there has to be at least c more positive than negative literals. On the one hand, we show that a random GP-tree with a linear number of leaves expresses any given variable with constant probability. On the other hand, with constant probability such a tree has a majority of negative literals of any given variable (indeed, there is a constant probability that the variable has neither positive nor negative literals in the GP-tree). This yields a plateau of equal fitness which can only be overcome by adding c positive literals, i.e., we need a rich set of neutral mutations that allow genetic drift to happen. Bloat control suppresses this genetic drift by biasing the search towards smaller solutions. Specifically, it may not allow to add positive literals one by one, which results in an infinite run time (see Lemma 1). Note that allowing the local search to add c leaves at the same time still results only in a small chance of $\mathrm{O}(n^{-c})$ of jumping the plateau. Hence, the $+c$-MAJORITY fitness function serves as an example where bloat control explicitly harms the search.

For 2/3-MAJORITY, a variable is expressed if its positive literals hold a 2/3 majority, i.e., if 2/3 of all its literals are positive. The fitness associated with 2/3-MAJORITY is the number of expressed variables while for 2/3-SUPERMAJORITY each expressed variable contributes a score between 1 and 2, where larger majorities give larger scores (see Section 2 for details). The variant 2/3-SUPERMAJORITY is utilized to aggravate the effect of bloat since it rewards large numbers of (positive) literals. We show that local search with bloat control is efficient for these two problems (Theorems 2 and 5). However, without bloat control local search fails on 2/3-SUPERMAJORITY due to bloat (see Theorem 6).

Regarding optimization without bloat control, we obtain experimental results as depicted in Figure 2. They provide a strong indicator that, when no bloat control is applied, optimization of $+c$-MAJORITY is efficient, in contrast to the case of bloat control. The trend for 2/3-MAJORITY indicates that optimization proceeds significantly more slowly without bloat control than with bloat control. Nevertheless, optimization seems to be feasible in contrast to the case of 2/3-SUPERMAJORITY.

Subsequently, we study a simple crossover which works as follows. The algorithm maintains a population of λ individuals, which are initialized randomly before a local search with bloat control is performed for a number of iterations. As a local search we employ the (1+1) GP, a simple mutation-only GP which iteratively either adds, deletes, or substitutes a vertex of the tree. We employ this algorithm for a number of rounds large enough to ensure that each vertex of the tree has been considered for deletion at least once with high probability, which aims at controlling bloat. Afterwards, the optimization proceeds in rounds; in each round, each individual t_0 is mated with a random other individual t_1 by joining t_0 and t_1 to obtain a tree t' which contains both t_0 and t_1; then local search is performed on t' as before yielding a tree t''. If t'' is at least as fit as t_0, we replace t_0 in the population by t''. The algorithm is called *Concatenation* since it joins two individuals, which is basically a concatenation. It is different from other approaches for memetic crossover GP as found, for example, in [4]. Note that this crossover is very different from GP crossovers found in the literature because of its almost complete disregard for the tree structure of the individuals.

However, this crossover already highlights some benefits which can be obtained with crossover, and it has the great advantage of being analyzable.

We show that the Concatenation Crossover GP with bloat control efficiently optimizes all three test functions $+c$-Majority, 2/3-Majority as well as 2/3-SuperMajority, due to its ability to combine good solutions (see Theorem 10). We summarize our findings in Table 1.

In Section 2 we state the formal definitions of algorithms and problems, as well as the mathematical tools we use. Section 3 gives the results for local search *with* bloat control, Section 4 for local search *without* bloat control and Section 5 for the Concatenation Crossover GP. In Section 6 we show and discuss our experimental results, before Section 7 concludes the paper.

Due to space restrictions, we only provide sketches for the proofs. A full version of the paper can be found at https://arxiv.org/abs/1805.10169.

2 Preliminaries

For a given n we let $[n] = \{1, \ldots, n\}$ be the set of variables. The only non-terminal (function symbol) is J of arity 2; the *terminal set* X consists of $2n$ literals, where $\overline{x}_i$ is the complement of x_i:

$$F \coloneqq \{J\},\ J \text{ has arity } 2, \qquad X \coloneqq \{x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n\}.$$

For a GP-tree t, we denote by $S(t)$ the set of leaves in t. By $S_i^+(t)$ and $S_i^-(t)$ we denote the set of leaves that are x_i-literals and $\overline{x}_i$-literals, respectively, and by $S_i(t) \coloneqq S_i^+(t) \cup S_i^-(t)$ we denote the set of all i-literals. By $S^+(t) \coloneqq \bigcup_{i=1}^n S_i^+(t)$ and $S^-(t) \coloneqq \bigcup_{i=1}^n S_i^-(t)$ we denote the set of all positive and negative leaves, respectively. We denote the sizes of all these sets by the corresponding lower case letters, i.e., $s(t) \coloneqq |S(t)|$, $s_i(t) \coloneqq |S_i(t)|$, etc.. In particular, we refer to $s(t)$ as the *size* of t.

On the syntax trees, we analyze the problems $+c$-Majority, 2/3-Majority, and 2/3-SuperMajority, which are defined as

$$+c\text{-Majority} \coloneqq |\{i \in [n] \mid s_i^+ \geq s_i^- + c\}| ;$$

$$2/3\text{-Majority} \coloneqq |\{i \in [n] \mid s_i \geq 1 \text{ and } s_i^+ \geq \tfrac{2}{3} s_i\}| ;$$

$$2/3\text{-SuperMajority} \coloneqq \sum_{i=1}^n f_i, \text{ where } f_i \coloneqq \begin{cases} 0 & \text{, if } s_i = 0 \text{ or } s_i^+ < \frac{2}{3} s_i, \\ 2 - 2^{s_i^- - s_i^+}, & \text{otherwise.} \end{cases}$$

We call a variable contributing to the fitness *expressed*. Since both $+c$-Majority and 2/3-Majority count the number of expressed variables, they take values between 0 and n. The function 2/3-SuperMajority is similar to 2/3-Majority, but if a 2/3 majority is reached 2/3-SuperMajority awards a bonus for larger majorities: the term f_i grows with the difference $s_i^+ - s_i^-$. Since $f_i \leq 2$, the function 2/3-SuperMajority takes values in $[0, 2n]$. Note that the value $2n$ can never actually be reached, but can be arbitrarily well approximated.

In this paper we consider simple mutation-based genetic programming algorithms which use a modified version of the Hierarchical Variable Length (HVL) operator ([21], [22]) called HVL-Prime as discussed in [3]. HVL-Prime allows trees of variable length to be produced by applying three different operations:

Given a GP-tree t, mutate t by applying HVL-Prime. For each application, choose uniformly at random one of the following three options.

substitute	Choose a leaf uniformly at random and substitute it with a leaf in X selected uniformly at random.
insert	Choose a node $v \in X$ and a leaf $u \in t$ uniformly at random. Substitute u with a join node J, whose children are u and v, with the order of the children chosen uniformly at random.
delete	Choose a leaf $u \in t$ uniformly at random. Let v be the sibling of u. Delete u and v and substitute their parent J by v.

Fig. 1: Mutation operator HVL-Prime.

insert, delete and substitute (see Figure 1). Each application of HVL-Prime chooses one of these three operations uniformly at random. We note that the literature also contains variants of the mutation operator that apply several such operations simultaneously (see [3, 20]).

The first algorithm we study is the (1+1) GP. The algorithm is initialized with a tree generated by s_{init} random insertions. Afterwards, it maintains the best-so-far individual t. In each round, it creates an offspring of t by mutation. This offspring is discarded if its fitness is worse than t, otherwise it replaces t. We recall that the fitness in the case with bloat control contains the size as a second order term. Algorithm 1 states the (1+1) GP more formally.

Algorithm 1: (1+1) GP with mutations according to Figure 1

```
1 Let $t$ be a random initial tree of size $s_{\text{init}}$;
2 **while** *optimum not reached* **do**
3     $t' \leftarrow \text{mutate}(t)$;
4     **if** $f(t') \geq f(t)$ **then** $t \leftarrow t'$;
```

2.1 Crossover

The second algorithm we consider is population-based. When introduced by J. R. Koza [12], Genetic Programming used fitness-proportionate selection and a genetic crossover, however mutation was hardly considered. In subsequent works many different setups for the crossover operator were introduced and studied. For instance, in [21] combinations of GP with local search in the form of mutation operators were studied and yielded better performance than GP.

Usually, two parents (a *current solution* and a *mate*) are used to generate a number of offspring. These offspring are a recombination of the alleles from both parents derived in a probabilistic manner. By modeling each individual as a GP-tree, a crossover-point in both parents is decided upon due to a heuristic and the subtrees attached to these points are exchanged creating new GP-trees.

In the Crossover hill climbing algorithm first described by T. Jones [7, 8] only one GP-tree is created from the current solution and a random mate. This offspring is evaluated and replaces the current solution if the fitness is not worse.

We consider the following simple crossover: the *Concatenation Crossover GP* working as follows (see also Algorithm 2). For a fixed population of GP-trees, each GP-tree is chosen to be the parent once. For each parent we choose a mate uniformly at random from the population and create one offspring by *joining* the two trees using a new join-node. Before evaluating the offspring, we employ a local search in the form of the (1+1) GP with bloat control. This local search is performed for a fixed amount of iterations before we discard the GP-tree with worse fitness. The fixed amount depends on the size of the tree and ensures the absence of redundant leaves with high probability (see Lemma 11). We note that the amount of redundant leaves depends on the function to be optimized. The functions we studied are variants of MAJORITY, for other functions the amount of iterations ensuring the absence of redundant leaves might be different.

The initial population is generated by creating λ random trees of size s_{init} and employing the local search on each of them. We then proceed in rounds of crossover as described above. We note that we assume all crossover operations to be performed in parallel. Hence, the new population is based entirely on the old population and not partially on previously generated individuals of the new generation.

Algorithm 2: Concatenation Crossover-GP

```
1  Let LS($t$) denote local search by the (1+1) GP with bloat control on tree $t$ for
   $90s \log s$ steps, where $s$ is the number of leaves in $t$;
2  for $i = 1$ to $\lambda$ do
3      Let $t_i$ be a random initial tree of size $s_{\text{init}}$;
4      $t_i \leftarrow \text{LS}(t_i)$;
5  while optimum not reached do
6      for $i = 1$ to $\lambda$ do
7          Choose $m \in \{1, \ldots, \lambda\} \setminus \{i\}$;
8          $t'_i \leftarrow \text{join}(t_i, t_m)$;
9          $t''_i \leftarrow \text{LS}(t'_i)$;
10         if $f(t''_i) \geq f(t_i)$ then $t_i \leftarrow t''_i$;
```

2.2 Terminology

For the analysis, it will be helpful to partition the set of leaves into three classes as follows. The set $C^+(t) \subseteq S^+(t)$ of *positive critical leaves* is the set of leaves u, whose deletion from the tree results in a decreased fitness. Similarly, the set $C^-(t) \subseteq S^-(t)$ of *negative critical leaves* is the set of leaves u, whose deletion from t results in an increased fitness. Finally, the set $R(t) := [n] \setminus (C^+(t) \cup C^-(t))$ of *redundant leaves* is the set of all leaves u, whose deletion from t does not affect the fitness. Similar as before, we denote $c^-(t) = |C^-(t)|$, $c^+(t) = |C^-(t)|$, and $r(t) = |R(t)|$.

Given a time $\tau \geq 0$, we denote by t_τ the GP-tree after τ iterations of the algorithm. Additionally, we use $S(\tau), s(\tau), S_i(\tau), \ldots$ in order to denote $S(t_\tau), s(t_\tau), S_i(t_\tau), \ldots$. Moreover, we apply the standard Landau notation $\mathrm{O}(\cdot)$, $\mathrm{o}(\cdot)$, $\Omega(\cdot)$, $\omega(\cdot)$, $\Theta(\cdot)$ as detailed in [1].

3 (1+1) GP with Bloat Control

In this section we study how local search with bloat control performs on the given fitness functions. Theorem 1 shows that for small initial trees $+c$-MAJORITY cannot be efficiently optimized, while Theorem 2 shows that this is possible for 2/3-MAJORITY. Finally, Theorem 5 considers 2/3-SUPERMAJORITY.

Theorem 1. *Consider the (1+1) GP on* $+c$-MAJORITY *with bloat control on the initial tree with size* $s_{\text{init}} < n$*. If* $c > 1$*, with probability equal to* 1*, the algorithm will never reach the optimum.*

The proof is based on an optimal GP-tree for $+c$-MAJORITY needing cn leaves, but bloat control does not allow to add leaves without fitness gain.

Next we state the upper bound for the performance on 2/3-MAJORITY. The proof of Theorem 2 is almost identical to the one of Theorem 4.1 in [2], the bounds stated in Lemma 4.2 and Lemma 4.1 in [2] need to be suitably adjusted, since these do not hold for 2/3-MAJORITY.

Theorem 2. *Consider the (1+1) GP on* 2/3-MAJORITY *with bloat control on the initial tree with size* s_{init}*. The expected time until the algorithm computes the optimum is in* $\mathrm{O}(n \log n + s_{\text{init}})$*.*

Corollary 3. *Consider the (1+1) GP on* 2/3-MAJORITY *with bloat control on the initial tree with size* $s_{\text{init}} < n$*. The expected time until the algorithm computes the optimum is in* $\mathrm{O}(n \log n)$*.*

We turn to 2/3-SUPERMAJORITY with Theorem 5. The proof is based on the following lemma showing that redundant leaves will be removed with sufficient probability. Hence, insertions of positive literals can increase fitness.

Lemma 4. *Consider the (1+1) GP on* 2/3-SUPERMAJORITY *with bloat control with* $n \geq 55$ *on the initial tree with size* $s_{\text{init}} < n$*. With probability at least* $1 - (\tau/(n \log^2 n))^{-1/(1+4/\sqrt{\log n})}$ *the algorithm will delete any given negative leaf of the initial tree within* $\tau \geq n \log^2 n$ *rounds. For a positive redundant leaf, with the same probability it will either be deleted or turned into a positive critical leaf.*

Theorem 5. *Consider the (1+1) GP on* 2/3-SUPERMAJORITY *with bloat control on an initial tree with size* $s_{\text{init}} < n$*, and let* $\varepsilon > 0$*. Then, the algorithm will express every literal after* $n^{2+\varepsilon}$ *iterations with probability* $1 - \mathrm{o}(1)$*.*

4 (1+1) GP without Bloat Control

In this section we study the fitness function 2/3-SUPERMAJORITY, which facilitates bloat of the string.

Theorem 6. *For any constant* $\nu > 0$*, consider the (1+1) GP without bloat control on* 2/3-SUPERMAJORITY *on the initial tree with size* $s_{\text{init}} = \nu n$*. There is* $\varepsilon = \varepsilon(\nu) > 0$ *such that, with probability* $1 - \mathrm{o}(1)$*, an* $\varepsilon-$*fraction of the indices will never be expressed. In particular, the algorithm will never reach a fitness larger than* $(2 - 2\varepsilon)n$*.*

We commence with some preparatory lemmas before proving the theorem. First, we analyze how the size of the GP-tree evolves over time. We recall that $s(\tau)$ is the number of leaves of the GP-tree at time τ.

Lemma 7. *There is a constant $0 < \eta \leq 1$ such that, with probability $1 - \mathrm{o}(1)$, for all $\tau \geq 0$ we have $s(\tau) \geq \eta\tau$.*

In order to continue we need some more terminology. For an index $i \in [n]$, we recall that $s_i^+(\tau)$ and $s_i^-(\tau)$ denote the number of x_i- and $\overline{x}_i$-literals at time τ, respectively, and $s_i(\tau) \coloneqq s_i^+(\tau) + s_i^-(\tau)$. We call index i *touched* in round τ, if a literal x_i or $\overline{x}_i$ is deleted, inserted or substituted, or if a literal is substituted by x_i or $\overline{x}_i$. We call the touch *increasing* if it is either an insertion or if a literal is substituted by x_i or $\overline{x}_i$. We call the touch *decreasing* if it is a deletion or substitution of a x_i or $\overline{x}_i$ literal. We note that in exceptional cases a substitution may be both increasing and decreasing. Let $\rho_i(\tau)$ be the number of increasing touches of i up to time τ. We call a decreasing step *critical* if it happens at time τ with $s_i(\tau) \leq \eta\tau/(4n)$, and we call $\gamma_i(\tau)$ the number of critical steps up to time τ. Finally, we call a round *accepting* if the offspring is accepted in this round.

The approach for the remainder of the proof is as follows. First, we will show that in the regime, where critical steps may happen (i.e, $s_i(\tau) \leq \eta\tau/(4n)$), it is more likely to observe increasing than decreasing steps. The reason is that a step is only critical if there are relatively few i-literals, in which case it is unlikely to delete or substitute one of them, whereas the probability to insert an i-literal is not affected. It will follow that $s_i(\tau)$ grows with τ, since otherwise we would need many critical steps. Finally, if $s_i(\tau)$ keeps growing it becomes increasingly unlikely to obtain a 2/3 majority. In order to state the first points more precisely we fix a $j_0 \in \mathbb{N}$ and call an index i *bad* (or more precisely, j_0-bad) if the following conditions hold: for all $\tau \geq j_0 n$ and $\tau_0 \coloneqq j_0 n$

(A) $s_i^+(\tau_0) \leq s_i^-(\tau_0) \leq j_0$ (B) $\tau/(2n) \leq \rho_i(\tau) \leq 2\tau/n$
(C) $\gamma_i(\tau) \leq 2\tau/n$ (D) $s_i(\tau) \geq \eta\tau/(8n)$.

In particular, in (A) x_i is not expressed at time τ_0.

Lemma 8. *For every fixed $i_0 > 0$, with probability $1 - \mathrm{o}(1)$ there are $\Omega(n)$ bad indices.*

Lemma 9. *Every bad index has probability $\Omega(1)$ that it is never expressed, independent of the other bad indices.*

We note that Lemmas 8 and 9 imply Theorem 6 by a straightforward application of the Chernoff bound.

5 Concatenation Crossover GP

In the following we will study the performance of the Concatenation Crossover GP (Algorithm 2) on $+c$-Majority and 2/3-Majority with bloat control. As observed in Theorem 1 the (1+1) GP with bloat control may never reach the optimum when optimizing an initial tree of size $s_{\text{init}} < n$. We will deduce that crossover solves this issue and the algorithm reaches the optimum fast. We commence this section by stating the exact formulation of the mentioned result in Theorem 10 followed by an outline of its proof. Finally, we show the corresponding result for 2/3-SuperMajority in Theorem 13.

Theorem 10. *Consider the Concatenation Crossover GP on* $+c$-Majority *or* 2/3-Majority *with bloat control on the initial tree with size* $2 \leq n/2 \leq s_{\text{init}} \leq bn$ *(for constant* $b > 0$*). Then there is a constant* $c_\lambda > 0$ *such that for all* $c_\lambda \log n \leq \lambda \leq n^2$*, with probability in* $(1 - \mathrm{O}(n^{-1}))$*, the algorithm reaches the optimum after at most* $\mathrm{O}(n \log^3(n))$ *steps.*

The following two auxiliary lemmas are used to proof the theorem. Here, they serve towards an outline of the proof. First, Lemma 11 states the absence of redundant leaves in a GP-tree t after the local search with a probability of $1 - n^{-5}$. This will be applied after every local search. We observe for two GP-trees t_1 and t_2 *without redundant leaves*: if t' is the tree resulting from joining t_1 and t_2, then a variable $i \in [n]$ is expressed in t' if and only if it is expressed in t_1 or t_2.

Second, Lemma 12 states that, with a probability of $1 - n^{-5}$, each variable $i \in [n]$ is expressed in at least one of $\lambda/2$ trees before the first crossover. Combining both lemmas, for a fixed GP-tree t it will suffice to observe the time until t has been joined with at least $\lambda/2$ different trees.

Lemma 11. *Consider the (1+1) GP with bloat control on either* $+c$-Majority *or* 2/3-Majority*. For an initial tree with size* $2 \leq n/2 \leq s_{\text{init}} \leq bn$ *(for constant* $b > 0$*) after* $90 s_{\text{init}} \log(s_{\text{init}})$ *iterations, with probability at least* $1 - n^{-5}$*, the current solution will have no redundant leaves.*

Lemma 12. *Consider the Concatenation Crossover GP on* $+c$-Majority *or* 2/3-Majority *with bloat control on initial trees with size* $2 \leq n/2 \leq s_{\text{init}} \leq bn$ *(for constant* $b > 0$*). Then there is a constant* $c_\lambda > 0$ *such that for all* $\lambda \geq c_\lambda \log n$*, with probability at least* $1 - n^{-5}$*, each variable will be expressed in at least one of* $\lambda/2$ *trees before the first crossover.*

Finally, we turn to 2/3-SuperMajority. For the proof we use a result from the area of rumor spreading relating to the *pull protocol* [9,17] in order to study the time until every individual of the population has every variable expressed. The idea here is similar to previous proofs with crossover: expressed variables can be collected with crossover. For this purpose we show that the number of x_i in individuals, which have a variable i expressed, is asymptotically larger than the number of $\overline{x}_i$ in individuals, which do not have i expressed.

Theorem 13. *Consider the Concatenation Crossover GP without substitutions with bloat control with initial tree size* $s_{\text{init}} = n/2$ *on* 2/3-SuperMajority*. Then there is a constant* $c_\lambda > 0$ *such that, for* $\lambda = c_\lambda \log n$*, each GP-tree in the population has all variables expressed after at most* $\mathrm{O}(n^{1+o(1)})$ *steps with probability at least* $1 - \mathrm{O}(n^{-4})$*.*

6 Experiments

This section is dedicated to complementing our theoretical results with experimental justification for the otherwise open cells of Table 1, i.e. for the (1+1) GP without bloat control on $+c$-Majority and 2/3-Majority.

All experimental results shown in Figure 2 are box-and-whiskers plots, where lower and upper whiskers are the minimal and maximal number of *fitness evaluations* the algorithm required over 100 runs until all variables are expressed or the

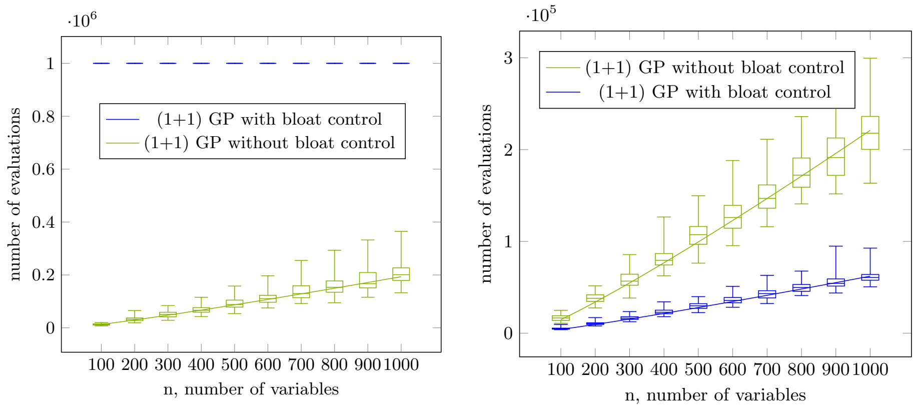

Fig. 2: Number of evaluations required by the (1+1) GP over 100 runs for each n with the initial tree size $s_{\text{init}} = 10n$ until all variables are expressed or the time limit, equal to 1000000 evaluations, is reached. The left figure shows the experimental results for $+c -$ Majority with $c = 2$; the solid line is $28n \log n$. On the right figure is shown $2/3 -$ Majority; the blue solid line is $9n \log n$, the green solid line is $32n \log n$.

time limit of 1000000 evaluations is reached. The middle lines in each box are the median values (the second quartile), the bottom and top of the boxes are the first and third quartiles. Note that all experiments are platform independent since we count number of fitness evaluations independently of real time. The solid lines in the plots allow to estimate the asymptotic run time of the (1+1) GP.

The left hand side of Figure 2 concerns $+c$-Majority and shows that the (1+1) GP with bloat control always fails (corresponding to Theorem 1). We used the (1+1) GP with $s_{\text{init}} = 10n$, $c = 2$ and n as indicated along the x-axis. It is easy to see that bloat control leads the algorithm to local optima and does not allow to leave it, whereas the (1+1) GP *without* bloat control finds an optimum in a reasonable number of evaluations. Due to time and computational restrictions the constant c was chosen equal to 2. For larger c the run time of the algorithm goes up significantly, but a similar pattern is visible.

The right hand side of Figure 2 shows the results of (1+1) GP on 2/3-Majority, using $s_{\text{init}} = 10n$. One can see that bloat control is more efficient in comparison with the (1+1) GP without bloat control. The set of median values is well-approximated by $w \cdot n \log n$ for a constant w, which leads us to the conjecture that the algorithm's run time is $O(n \log n)$. We did not analyze the influence of s_{init}, but it might be significant especially for 2/3-Majority without bloat control.

7 Conclusion

We defined three variants of the Majority problem in order to introduce some fitness plateaus that are difficult to cross. The $+c$-Majority allows for progress at the end of the plateau with large representation; in this sense, bloat is necessary for progress. On the other hand, for 2/3-Majority, progress can be made at the end of the plateau with small representation, so that bloat control guides the search to the fruitful part of the search space. We also considered 2/3-SuperMajority which exemplifies fitness functions where bloat is inherent due to the possibility of small improvements by adding an increasing amount of nodes to the GP-tree. In this case we showed that not employing bloat control leads to inefficient optimization.

In order to obtain results somewhat closer to practically relevant GP we turned to crossover and showed how a Concatenation Crossover GP can efficiently optimize all three considered test functions.

For future work it might be interesting to analyze the effect of other crossover operators. In order to obtain a better understanding of such other operators, other test functions might be necessary making essential use of the tree structure (all our test functions might as well use lists or even multisets of the leaves as representations). Such test functions should not be too complex, which would hinder a theoretical analysis, but still embody a structure frequently found in GP, so as to inform about relevant application areas. The search for such test functions remains a central open problem of the theory of GP.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, 2. edn. (2001)
2. Doerr, B., Kötzing, T., Lagodzinski, J.A.G., Lengler, J.: Bounding bloat in genetic programming. In: Proc. of GECCO'17. pp. 921–928. ACM (2017)
3. Durrett, G., Neumann, F., O'Reilly, U.M.: Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics. In: Proc. of FOGA'11. pp. 69–80 (2011)
4. Eskridge, B.E., Hougen, D.F.: Memetic crossover for genetic programming: Evolution through imitation. In: Proc. of GECCO'04. pp. 459–470 (2004)
5. Gathercole, C., Ross, P.: An adverse interaction between the crossover operator and a restriction on tree depth. In: Proc. of GP'96. pp. 291–296 (1996)
6. Goldberg, D.E., O'Reilly, U.M.: Where does the good stuff go, and why? How contextual semantics influences program structure in simple genetic programming. In: Proc. of EuroGP'98. pp. 16–36 (1998)
7. Jones, T.: Crossover, macromutation, and population-based search. In: Proc. of ICGA'95. pp. 73–80. Morgan Kaufmann Publishers Inc. (1995)
8. Jones, T.: Evolutionary Algorithms, Fitness Landscape and Search. Ph.D. thesis, University of New Mexico (1995)
9. Karp, R.M., Schindelhauer, C., Shenker, S., Vöcking, B.: Randomized rumor spreading. In: Proc. of FOCS'00. pp. 565–574 (2000)
10. Kötzing, T., Neumann, F., Spöhel, R.: PAC learning and genetic programming. In: Proc. of GECCO'11. pp. 2091–2096 (2011)
11. Kötzing, T., Sutton, A.M., Neumann, F., O'Reilly, U.M.: The Max problem revisited: the importance of mutation in genetic programming. In: Proc. of GECCO'12. pp. 1333–1340 (2012)
12. Koza, J.R.: Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Tech. rep., Stanford, CA, USA (1990)
13. Langdon, W.B., Poli, R.: An analysis of the MAX problem in genetic programming. In: Proc. of GP'97. pp. 222–230 (1997)

14. Luke, S., Panait, L.: Lexicographic parsimony pressure. In: Proc. of GECCO'02. pp. 829–836 (2002)
15. Mambrini, A., Manzoni, L.: A comparison between geometric semantic GP and cartesian GP for Boolean functions learning. In: Proc. of GECCO'14. pp. 143–144 (2014)
16. Mambrini, A., Oliveto, P.S.: On the analysis of simple genetic programming for evolving Boolean functions. In: Proc. of EuroGP'16. pp. 99–114 (2016)
17. Mercier, H., Hayez, L., Matos, M.: Optimal epidemic dissemination. CoRR abs/1709.00198 (2017), http://arxiv.org/abs/1709.00198
18. Moraglio, A., Mambrini, A., Manzoni, L.: Runtime analysis of mutation-based geometric semantic genetic programming on Boolean functions. In: Proc. of FOGA'13. pp. 119–132 (2013)
19. Neumann, F.: Computational complexity analysis of multi-objective genetic programming. In: Proc. of GECCO'12. pp. 799–806 (2012)
20. Nguyen, A., Urli, T., Wagner, M.: Single- and multi-objective genetic programming: new bounds for weighted ORDER and MAJORITY. In: Proc. of FOGA'13. pp. 161–172 (2013)
21. O'Reilly, U.M.: An Analysis of Genetic Programming. Ph.D. thesis, Carleton University, Ottawa, Canada (1995)
22. O'Reilly, U.M., Oppacher, F.: Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In: Proc. of PPSN'94. pp. 397–406 (1994)