The *Max* problem revisited: The importance of mutation in genetic programming

Timo Kötzing [a], Andrew M. Sutton [b,*], Frank Neumann [b], Una-May O'Reilly [c]

[a] *Department of Algorithms and Complexity, Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany*
[b] *Evolutionary Computation Group, School of Computer Science, The University of Adelaide, Adelaide, SA 5005, Australia*
[c] *MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA*

ARTICLE INFO

ABSTRACT

We study the importance of mutation in genetic programming and contribute to the rigorous understanding of genetic programming algorithms by providing runtime complexity analyses for the well-known *Max* problem. Several experimental studies have indicated that it is hard to solve the *Max* problem with crossover-based algorithms. Our analyses show that different variants of the *Max* problem can provably be solved efficiently using simple mutation-based genetic programming algorithms.

Our results advance the body of computational complexity analyses of genetic programming, indicate the importance of mutation in genetic programming, and reveal new insights into the behavior of mutation-based genetic programming algorithms.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Genetic programming (GP) is a class of evolutionary algorithms that evolve, for a specific task, executable structures, such as computer programs. Pioneered by Koza [10] in the early 1990s, genetic programming has been demonstrably successful at solving human competitive programming problems arising from diverse domains.

The goal of this paper is to contribute to the growing field of computational complexity analysis for genetic programming. This type of analysis has significantly increased the theoretical understanding of other types of evolutionary algorithms (see the books [1,14] for a comprehensive presentation).

Our objective is to examine the importance of mutation in genetic programming in a rigorous way. There are numerous examples of genetic programming where subtree crossover is the only variation operator used. The *Max* problem [5], defined formally in Section 2, was initially introduced as a means of qualitatively gauging the limitations of crossover as it interplays with a fixed tree height (or size). It has a very nice and simple formulation and is easy to solve analytically. Given a set of functions, a set of terminals, and a bound D on the maximum depth of a genetic programming tree, the goal is to evolve a tree that returns the maximum value possible given any combination of functions and terminals. It was observed that on the *Max* problem, "Subtree discovery and movement takes place mostly near the leaf nodes, with nodes near the root left untouched, where diversity drops quickly to zero in the tree population. GP is then unable to create fitter trees via the crossover operator, leaving a mutation operator as the only common, but ineffective, route to discovery of fitter trees" [5, abstract]. A later investigation found that mutation would eventually help find an optimal solution, albeit slowly for the *Max* problem, i.e., "the later stages of GP runs are effectively performing randomized hill climbing and so solution time grows exponentially with depth of the solution" [11, p. 229].

* Corresponding author.
  E-mail address: sutton@cs.colostate.edu (A.M. Sutton).

We revisit the *Max* problem also because its solution space is easy to think about and its solutions are known in advance. Our algorithms employ the HVL-Prime operator introduced in [3]. HVL-Prime (defined formally in Section 2) is an update of one of the first GP mutation operators. These were either random subtree selection and replacement or substitution of a node with a different function or terminal (considering what arity the node is). HVL-Prime introduces more gentle and incremental variation by changing a program tree at just one node, or by growing or shrinking the tree by the minimum number of nodes possible. It satisfies the need to search trees that vary in both height, size and structure while introducing variations in these dimensions by the smallest step possible.

Initial steps in the computational complexity analysis of genetic programming have been made in [3] by studying the runtime of $(1 + 1)$ GP algorithms on the problems ORDER and MAJORITY introduced in [6]. These investigations have been extended in [13] to incorporate simple bloat-control, either directly by adjusting the fitness function or by multi-objective variants. Furthermore, GP algorithms have been studied in the PAC learning framework [8] and general studies on the learnability of evolutionary algorithms for Boolean functions have already been carried out before in [4,17]. ORDER and MAJORITY are in some sense easy to optimize because they have independent problem semantics. The $(1 + 1)$ GP algorithms studied in [3] use a variable-length representation that is important since syntax trees can usually grow and shrink during the optimization process. The inner nodes of such a tree contain function symbols, and leaf nodes represent terminal symbols, i.e. constants or variables. In ORDER and MAJORITY the only function is a join operation, which does not have any effect on the function value. In contrast, the *Max* problem has the important property that different functions such as addition and multiplication are available. However, the set of terminals is as simple as possible, i.e., it consists of only one specific constant. We will analyze variants of $(1 + 1)$ GP on different variations of the *Max* problem. The problems are different from each other only in the set of functions and terminals available for evolution of maximum solutions.

On this occasion, rather than using *Max* as a vehicle for qualitatively analyzing crossover, we derive the computational complexity of mutation-based hill climbing genetic programming algorithms that provably solve it. Some genetic programming algorithms have focused on using mutation as an alternative to crossover, to exploit its effectiveness in adaptive search. Stochastic iterated hill climbing (SIHC) [15,16] is one such example. SIHC is similar to our $(1 + 1)$ mutation-based hill-climber. For a search process bounded by a maximally sized tree of n nodes, the time complexity of our algorithms for the entire range of variants is bounded by $O(n \log n)$ when one mutation operation precedes each fitness evaluation. When multiple mutations are successively applied before each fitness evaluation, the time complexity bound is $O(n^2)$.

We investigate the mutation operator further by an analysis of the growth process of binary trees. We show that the expected time to create particular nodes is infinite when there is no limit on the maximum depth. This negative result can be avoided if the mutations are biased to replace a random leaf with distance d from the root with probability 2^{-d}. We design a biased mutation operator based on these insights and show that they also lead to an expected optimization time of $O(n \log n)$ when using single mutation steps. Furthermore, we show by experimental investigations that the biased mutation operator outperforms the standard mutation-based approach by a factor of up to six.

These polynomial time bounds for a simple hill climbing algorithm contrast previous empirical research that suggests subtree crossover in a population-based setting is not effective on the *Max* problem. Our intention is not to question crossover and we do not claim that genetic programming problems are all well-represented by *Max*. However, our analyses suggest that there is low cost and possible value to evaluating the scale at which a mutation-based hill-climber effectively solves a given problem before it is necessary to resort to a population- and crossover-based genetic programming algorithm. They further suggest that, with informed guidance, depth-dependent mutation can effectively support the exploration and discovery of solution-dependent genetic materials.

This article is an extension of the conference publication [9]. Compared to the conference version, the runtime bounds when using single mutation operations are improved from $O(n \log^2 n)$ to $O(n \log n)$. In addition, the results hold for all positive terminal values instead of just the special cases considered in [9]. This holds for the standard mutation operator and the biased mutation operator. Furthermore, this article complements the theoretical investigations by experimental studies. In particular, our experimental investigations show the advantage of using biased mutations for the *Max* problem.

We proceed as follows. In Section 2, we formally introduce the problems and algorithms we investigate. In Section 3, we study $(1 + 1)$ GP-single where mutation is applied once before each fitness evaluation. Our strategy is to consider which operations monotonically increase the fitness of the current solution so we can compute the likelihood of each layer of the tree becoming fixed to correct functions and the overall cost of fixing every level correctly. The expected time to solve the problem is dominated by the time to fix the tree at its deepest level, D. We present results for the more general $(1 + 1)$ GP-multi in Section 4. Finally, we show in Section 5 how biased mutation operators can lead to a better growth process and present our experimental investigations in Section 5.3. We finish with some discussion and conclusions.

2. The *Max* problem

The task for the *Max* problem is to find a program (modeled by a syntax tree) that returns the largest possible value for a given set of binary functions F, terminal set T, with a depth limit D for the syntax tree. The complete tree of depth D has therefore 2^D leaves, $2^D - 1$ inner nodes, and in total $2^{D+1} - 1$ nodes. We write $n = 2^{D+1} - 1$ to denote the maximal number of nodes in a binary tree of depth D; this n is what we consider the problem size that we will use in our runtime bounds.

We investigate different variants of the *Max* problem as defined in [5]. This problem has already been subject to theoretical investigations of genetic programming using only crossover as a variation operator [11], which show that such algorithms find it difficult to produce optimal solutions.

For fixed D, F, T, we denote the corresponding *Max* problem as MAX-depth-D-F-T. We consider the following types of the *Max* problem.

- The problem MAX-depth-D-$\{+\}$-$\{t\}$, $t > 0$ a positive constant, is the perhaps simplest *Max* problem; the optimal tree is a full binary tree, with $+$ at interior nodes and ts at leaves. The value of an optimal solution is $t \cdot 2^D$.
- MAX-depth-D-$\{+, \times\}$-$\{t\}$, $t > 0$ a positive constant, is the problem where we have two functions $+$ and $\times$ and one terminal symbol t. Notable special cases for t include the following.
  - If $t > 2$, then, for all interior nodes, $\times$ is always preferred over $+$.
  - If $t = 1$, then we can have large subtrees evaluating to 1. For any given node, as long as one of its child trees evaluates to 1, a label of $+$ is preferable; if both child trees evaluate to something $\geqslant 2$, a label of $\times$ is preferred.
  - If $0 < t < 1$, then there are a number of layers of $+$ (counting from just above the leaves) before $\times$ is strictly preferred.

The *Max* problem is like a typical GP problem in the way it requires a function to be evolved. Whereas a typical GP problem uses terminal symbols for variables, without loss of generality, terminal symbols in the *Max* problem are only constants. Where our analysis mentions integer terminal symbols, variables of the same range could be substituted. The use of explicit constants allows the direct calculation of fitness (in lieu of executing a training set) and does not impact of our theoretical analysis.

For a given tree X, we will use $F(X)$ to denote the number of inner nodes (labeled with functions) and by $T(X)$ the number of terminals (leaves). Note that $T(X) = F(X) + 1$ holds for any possible tree X. The fitness $f(X)$ of a tree is the value at the root node, in other words, the value computed by the entire syntax tree, if the depth of X is at most D and 0 if the depth exceeds D.

We consider the HVL-Prime operator composed of the three following mutation operators on a given tree X.

- The operator `substitute` replaces a randomly chosen inner node of X with a new node $u \in F$ selected uniformly at random.
- The operator `insert` randomly chooses a leaf v in X and selects u uniformly at random from T; then it replaces v with a node w chosen uniformly at random from F whose children are u and v, with the order of the children chosen randomly.
- The operator `delete` randomly chooses a leaf node v of X, with parent p and sibling u; then it replaces p with u and deletes p and v.

We analyze a $(1+1)$ GP algorithm without crossover (detailed in Algorithm 1 below) that performs k HVL-Prime mutation operations to produce an offspring. We consider two variants that differ by how they choose k. For $(1+1)$ GP-single, we set $k = 1$, so that it performs exactly one mutation at a time according to the HVL-Prime framework. For $(1+1)$ GP-multi, we choose $k = 1 + \text{Pois}(1)$, so that the number of mutations at a time varies randomly according to the Poisson distribution with parameter 1. Note that this means that the probability of choosing any fixed k does not depend on the problem size n, and we will implicitly use the observation that k following this distribution is, with constant probability, equal to 1 and, similarly, it is with constant probability equal to 2.

Algorithm 1: $(1+1)$ GP.

---
1 Create a single-node tree X by choosing a terminal from T u.a.r.;
2 **while** *optimum not reached* **do**
3 $X' \leftarrow X$;
4   Choose k;
5   **for** $i = 1$ **to** k *times* **do**
6   Choose $m \in \{\texttt{substitute}, \texttt{insert}, \texttt{delete}\}$ u.a.r.;
7 $X' \leftarrow m(X')$;
8   **if** $f(X') \geqslant f(X)$ **then** $X \leftarrow X'$;

---

3. Analysis for $(1+1)$ GP-single

We begin by analyzing the simple case of the $(1+1)$ GP-single in which a single mutation is performed in each iteration. For all versions we consider, we show that the $(1+1)$ GP-single can efficiently solve the *Max* problem in time bounded above by $O(n \log n)$. We first introduce some definitions that facilitate the analysis.

Let $\mathscr{T} = \mathscr{T}(D, F, T)$ be the set of all (ordered) full binary trees of maximum depth D constructible from interior nodes from F and leaves from T, i.e., the set of all feasible candidate solutions for MAX-depth-D-F-T.

We recursively label each node of a full ordered binary tree $X \in \mathscr{T}$ as follows. The root of X is labeled 1. For any node $v \in X$ labeled a, the left child of v is labeled $2a$, and the right child is labeled $2a + 1$. Let $a \in \{1, \ldots, 2^{D+1} - 1\}$. We define the indicator function $\xi_a : \mathscr{T} \to \{0, 1\}$ as

$$\xi_a(X) = \begin{cases} 1 & \text{if there is a node in } X \text{ labeled } a; \\ 0 & \text{otherwise.} \end{cases}$$

Note that the label of a node in a tree X corresponds to its position in an embedding in the complete binary tree of depth D. We will therefore subsequently refer to a node's label as its position in the complete tree.

We consider the execution of the $(1 + 1)$ GP-single on MAX-depth-D-F-T as a stochastic process $\{X_i : i \in \mathbb{N}\}$ where X_i is a $\mathscr{T}$-valued random variable on an appropriate probability space. Intuitively, X_i represents the tree constructed by the $(1 + 1)$ GP-single in the i-th iteration.

Definition 1. Let $\{X_i : i \in \mathbb{N}\}$ be the stochastic process associated with the execution of the $(1 + 1)$ GP-single on MAX-depth-D-F-T. We say a position a is *fixed* for X_i if

$$\Pr\bigl(\xi_a(X_j) = 1 \mid X_i\bigr) = 1$$

for all X_j with $j \geqslant i$. A position that is not fixed is called *unfixed*.

Intuitively, a position is fixed once no sequence of legal operations by the $(1 + 1)$ GP-single result in that position becoming empty. We will sometimes state that the $(1+1)$ GP-single *fixes* a position a in iteration i. This corresponds to the event that a is unfixed in X_{i-1} but fixed in X_i.

Definition 2. We say a tree is *fixed to level* $0 \leqslant k \leqslant D$ if every position at depth up to k is already fixed for that tree.

We now show that, as long as the probability of fixing a position is not too small, the $(1 + 1)$ GP-single can efficiently fix all positions down to a particular level in the tree using only the mutation operator described in Section 2. We will later prove this probability bound holds for all cases we consider.

Lemma 3. *During the execution of $(1 + 1)$ GP-single, if the probability to fix an unfixed child of a fixed position is $\Omega(1/n)$, then the expected time until the $(1 + 1)$ GP-single produces a tree that is fixed to level $m \leqslant \log n$ is $O(n \log n)$.*

Proof. We consider all positions corresponding to the leaves of the complete binary tree of depth m. Consider a single leaf position v. There are at most m ancestors of v in the tree (starting at the root) that need to be fixed before v itself can be fixed. Assuming at least one ancestor of v is already fixed (e.g., the root), the probability of increasing the number of fixed ancestors of v is $\Omega(1/n)$.

Thus the time until every leaf position is fixed is the total time it takes to increase the number of fixed ancestors of each leaf position until there are no unfixed positions on the path from any leaf position to the root. Since the number of unfixed positions in a path from any leaf position to the root is bounded above by m, the time it takes to fix all the leaf positions is stochastically dominated by the time to collect $n \geqslant 2^m$ coupons (there are 2^m leaf positions), each at least m times.

From [7] we know that the time to collect n coupons each m times (assuming that in each iteration one coupon is received uniformly at random) is at most

$$O\bigl(n(\log n + m \log \log n - \log m!)\bigr).$$

Using Stirling's approximation for $\log m!$ and using $m = O(\log n)$, we get that this is $O(n \log n)$ as desired. □

We are now ready to prove bounds on the runtime of the $(1 + 1)$ GP-single on different variants of the *Max* problem. We start with the simplest variant, MAX-depth-D-$\{+\}$-$\{t\}$. See Fig. 1 for an illustration of the optimal tree for $D = 4$ on this variant.

Theorem 4. *For any $t > 0$, the expected time for the $(1 + 1)$ GP-single to optimize MAX-depth-D-$\{+\}$-$\{t\}$ is in $O(n \log n)$.*

Proof. During execution of the $(1 + 1)$ GP-single, a deletion is never accepted since it would always result in an inferior solution. Hence a position is fixed as soon as it contains a node. Moreover, the optimal solution is found as soon as the tree is fixed to level D, that is, the complete binary tree of depth D has been built.

An unfixed position with a fixed parent becomes fixed after a specific insertion operation occurring with probability at least $1/(3n)$. This satisfies the conditions of Lemma 3 and the expected time to fix the tree to level D is $O(n \log n)$. □
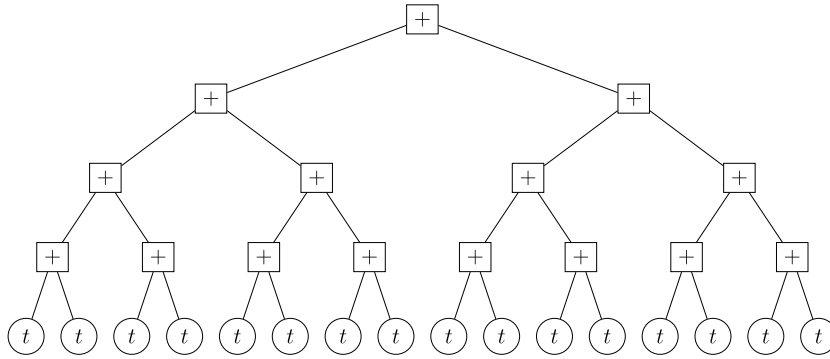
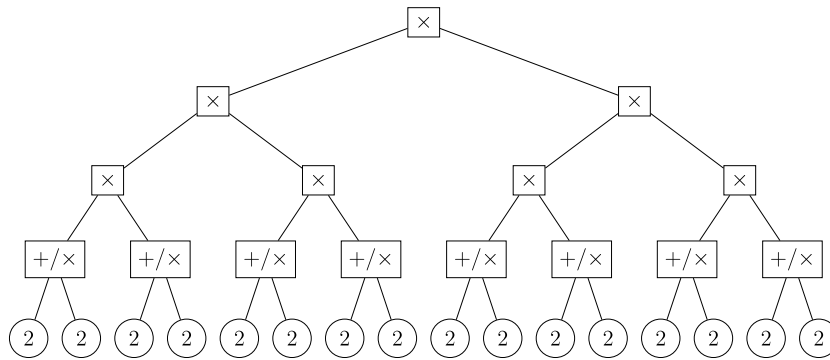**Fig. 1.** An optimal tree for MAX-depth-D-$\{+\}$-$\{t\}$ with $D = 4$.

Fig. 2. An optimal tree for MAX-depth-D-$\{+, \times\}$-$\{2\}$ with $D = 4$.

We are now interested in incorporating multiplication function nodes into the problem. The analysis for $F = \{+, \times\}$ is similar to the above case when the terminal value is strictly larger than one. As an example of this class of program tree, see Fig. 2 for the optimal depth-four tree when $t = 2$.

Theorem 5. *For all $t > 1$, the expected time for the $(1 + 1)$ GP-single to optimize MAX-depth-D-$\{+, \times\}$-$\{t\}$ is in $O(n \log n)$.*

Proof. Again, in this case, deletions are never accepted. Hence a position is fixed as soon as it contains a node. An unfixed position with a fixed parent becomes fixed after a specific insertion that occurs with probability at least $1/(3n)$. By Lemma 3, the tree becomes fixed to level D after $O(n \log n)$ operations in expectation.

After this point, the tree may not yet be optimal. If $t > 2$, the tree is optimal after all function nodes are transformed to $\times$ nodes. In this case, since $z^2 > 2z$ for all $z \geqslant t$, such a transformation always results in an improving fitness and by the coupon collector theorem [12], all nodes have been transformed to $\times$ nodes after $O(n \log n)$ operations in expectation. If $t = 2$, the function nodes at level $D - 1$ are irrelevant since $t^2 = 2t$ and the above bound remains valid.

If $1 < t < 2$, the tree is optimal when function nodes at depth $D - 1$ are $+$ nodes since in this case $t^2 < 2t$, and the remainder of the function nodes in the tree are $\times$ nodes. In this case, $z^2 > 2z$ for all $z \geqslant 2t$, so each specific substitution guarantees an improving move. Again by the coupon collector theorem, the correct substitutions have been made after $O(n \log n)$ operations in expectation. □

We now consider the case when the terminal values are constrained to exactly one. This case is slightly trickier because now it is possible that some deletion mutations are accepted.

Theorem 6. *The expected time for the $(1 + 1)$ GP-single to optimize MAX-depth-D-$\{+, \times\}$-$\{1\}$ is in $O(n \log n)$.*

Proof. The optimal solution is a complete binary tree of depth D where level D consists of 2^D leaves, level $D - 1$ consists of 2^{D-1} $+$ nodes, level $D - 2$ consists of 2^{D-2} $+$ or $\times$ nodes, and all remaining levels consist only of $\times$ nodes.

Unlike in the cases of MAX-depth-D-$\{+\}$-$\{1\}$ and MAX-depth-D-$\{+, \times\}$-$\{t\}$ for $t > 1$, a node can be deleted if it evaluates to 1 and its parent is a $\times$ node. However, if a function node evaluates to at least 2, it cannot be deleted since doing so would result in an inferior solution. In this case, a depth-D tree $X^\star$ is an optimal solution if and only if it is fixed to level D and all nodes at levels up to $D - 3$ are $\times$ nodes (see Fig. 3).
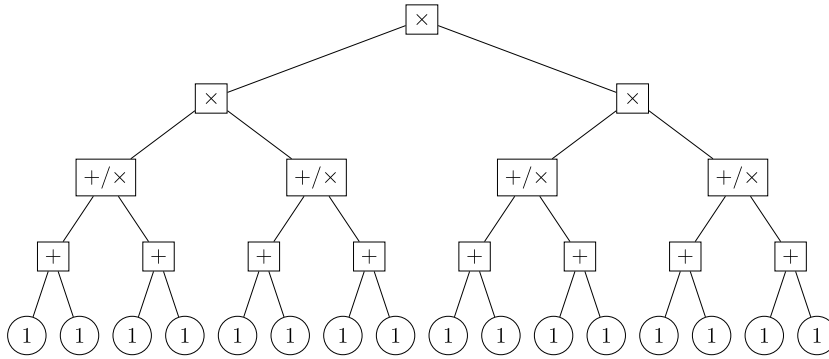
Fig. 3. An optimal tree for MAX-depth-*D*-{+, ×}-{1} with *D* = 4.

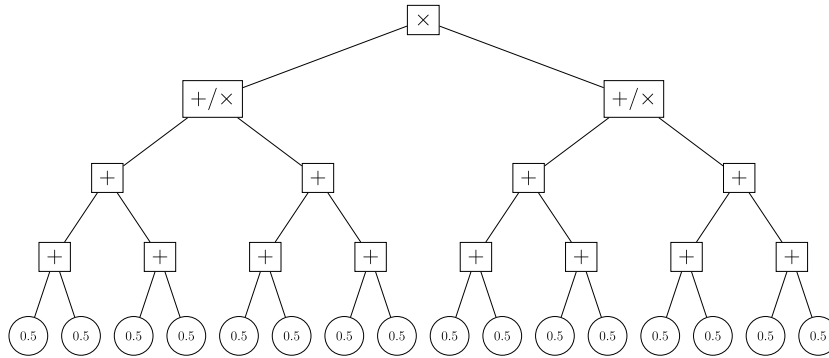**Fig. 4.** An optimal tree for MAX-depth-*D*-{+, ×}-{0.5} with *D* = 4.

Consider a node *v* in an unfixed position such that the parent of *v* is fixed. For this variant of the *Max* problem, a position containing a + node is always fixed. Hence, the position containing *v* becomes fixed if either (1) *v* is a × node that undergoes substitution to a + node, or (2) *v* is a leaf node which undergoes an insert operation involving a + node. Either of these situations occur with probability at least $1/(3n)$ so we can invoke Lemma 3 and the time until the tree becomes fixed to level *D* is $O(n \log n)$.

After this point, no deletions can occur, and no nodes at level $D - 1$ will be substituted to × nodes since this would result in an inferior tree. We can ignore any substitutions at level $D - 2$ since they result in a tree with the same value. The tree becomes optimal when all nodes above $D - 3$ are substituted into times nodes. A substitution of an arbitrary node occurs with probability at least $1/(3n)$. By the coupon collector theorem, all nodes are transformed to × nodes after an expected $O(n \log n)$ number of evaluations. □

3.1. Terminals of small value

Gathercole and Ross [5] suggested that the *Max* problem can be made progressively more difficult for traditional GP based on subtree crossover by decreasing the size of the constant terminal. For example, in their paper they used the values $t = 0.5$ and $t = 0.25$, and Fig. 4 illustrates the depth-four optimal tree for when $t = 0.5$. In this case, an optimal tree contains nodes labeled + on lower levels of the tree, and nodes labeled × higher in the tree.

On the other hand, we find that the asymptotic runtime of the $(1 + 1)$ GP-single is invariant under changes in terminal value. This is because the processes of building up the tree and collecting the correct internal nodes do not depend on *t*, even when it is made arbitrarily small. We first demonstrate this empirically in Fig. 5. Here we give the mean of the number of iterations for 10 000 independent runs for various *t* (standard deviation is indicated), and two values of *D* (5 and 8).

These results suggest that the time to create an optimal tree is independent of *t*, and the following theorem states that the runtime of the $(1 + 1)$ GP-single is the same for small values of *t* as it is for large values of *t*.

Theorem 7. *For any t, $0 < t < 1$, the expected time for the $(1 + 1)$ GP-single to optimize MAX-depth-D-{+, ×}-{t} is in $O(n \log n)$* (*asymptotic independence of t*).

Before proving Theorem 7, we make a few observations for the case of $0 < t < 1$ that facilitate the proof. Let us first define a *light node* as any interior node labeled × with at least one child evaluating to at most 1.
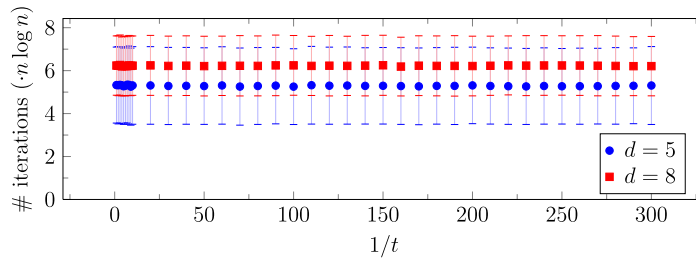
Fig. 5. Runtime for $(1+1)$ GP-single to optimize MAX-depth-D-$\{+, \times\}$-$\{t\}$.

Lemma 8. *Suppose $0 < t < 1$. Starting from any initial tree, after an expected $O(n \log n)$ iterations, the $(1+1)$ GP-single never generates a tree containing light nodes.*

Proof. We first observe that no new light nodes can be created. In particular, if a function node v is not a light node, it cannot be transformed into a light node by an accepted operation. If v is labeled $\times$, then if it is changed to a light node by an operation, one of its children must now evaluate to a strictly smaller value, resulting in a smaller fitness for the entire tree. Such an operation is not accepted by the $(1+1)$ GP-single.

On the other hand, if v is labeled $+$, the only way it can become light is if a substitute operation occurs at v and at least one of its children already evaluates to at most 1. Suppose the children of v evaluate to x and y. Without loss of generality let $x \leqslant y$. For v to become light, either $x \leqslant y \leqslant 1$ or $x \leqslant 1 < y$. In either case, $x + y > xy$ so the substitution would result in a strictly smaller value at v, and hence a tree with a smaller fitness value. Finally, since $t^2 < t$, an insertion operation that adds a new function node labeled $\times$ is not accepted. It follows that no operations that create new light nodes are ever accepted.

Now suppose we start with an initial tree that contains $k > 0$ light nodes. Let v be a particular light node whose children evaluate to x and y where $x \leqslant 1$. A substitution at v is always accepted since $xy \leqslant y < y + x$. Furthermore, the substitution removes that particular light node. Removing any light node by substitution occurs with probability at least $k/(3n)$. Since no new light nodes are created, the expected time until all light nodes have vanished is at most

$$3n \sum_{i=1}^{k} 1/i = O(n \log n).$$

Note that some light nodes can also be removed by deletion (if they have a leaf child), but this only means the light nodes vanish faster by at most a constant factor. $\square$

If the initial tree is a single leaf node, then light nodes can never appear during the execution of the $(1+1)$ GP-single. However, we would also like to show that the bound holds regardless of how the tree is initialized. The important matter is that a deletion can only occur when the parent of the deleted leaf is a light node, hence if there are no light nodes, there are no deletions possible.

Proof of Theorem 7. By Lemma 8, after $O(n \log n)$ time, all nodes are non-light. It follows that at this point, and for the remainder of the execution, no deletions are possible and, by an argument analogous to the proof of Theorem 4, the entire tree is fixed after at most $O(n \log n)$ expected iterations. Furthermore, after fixation, the interior node labels are corrected by collecting the needed substitution operations within the stated time bound. $\square$

4. Analysis for $(1+1)$ GP-multi

Theorem 9. *The expected time for the $(1+1)$ GP-multi to optimize MAX-depth-D-$\{+\}$-$\{1\}$ is in $O(n^2)$.*

Proof. Let X be a non-optimal tree, i.e., not the complete binary tree of depth D. Then there is at least one leaf in the tree that does not have depth D. A mutation that performs exactly one insertion operation at that leaf increases the fitness by 1. Such a mutation occurs with probability at least $1/(3en)$ since the probability that Poisson mutation performs exactly one operation is $1/e$ and there are 3 distinct types of operation. The fitness of a solution is determined by the number of leaves in the tree and is at most 2^D. Hence, the expected time until an optimal tree has been achieved is upper bounded by

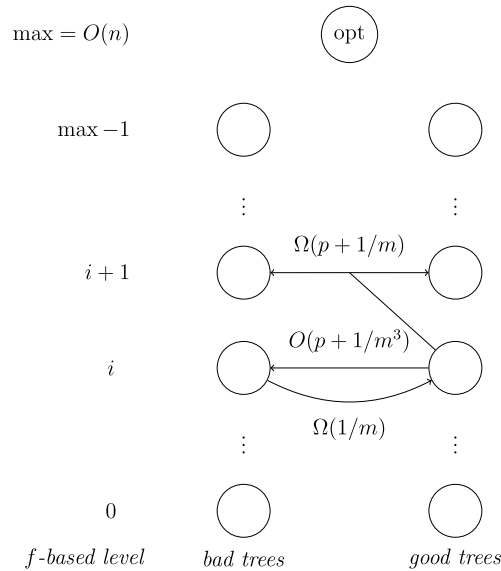$$\sum_{i=0}^{2^D-1} 3en = O(n^2). \qquad \square$$

Fig. 6. State diagram and transition probabilities between different types (good/bad) of tree and different f-based levels dependent on the number m of nodes in the tree.

In the theorem that follows, we bound the runtime of the $(1+1)$ GP-multi on MAX-depth-D-$\{+, \times\}$-$\{1\}$. The difficulty of the analysis lies in the variable number of operations: while $(1+1)$ GP-single makes exactly one application of the HVL-Prime operator, $(1+1)$ GP-multi makes a random number of them, as given by sampling a Poisson distribution with parameter 1; the *expected* number of applications of the HVL-Prime operator in each iteration is thus 2.

Theorem 10. *The expected time for the* $(1+1)$ *GP-multi to optimize MAX-depth-D-$\{+, \times\}$-$\{1\}$ is in $O(n^2)$.*

Proof. We argue by analyzing the Markov chain generated by the $(1+1)$ GP-multi . We divide the set of all possible trees (i.e., possible best-so-far solutions) into groups. Each tree is either *good* or *bad* as defined later in this proof; intuitively, *good* trees are easy to improve, while *bad* trees are not. We also divide the best-so-far solutions into f-based levels: given the value $f(X)$ it returns its f-based level is $\log_{6/5}(f(X))$. As an exception, the tree with optimal fitness has its own f-based level. We will show that good trees have a reasonable chance of improving their fitness by a factor of at least $6/5$, thus climbing up a level; furthermore, we will see that we will not have *bad* trees for too much of the time.

As the optimal fitness is $O(2^n)$, we have $\Theta(n)$ f-based levels, and up to 2 groups of trees per f-based level (the group of good ones and the group of bad ones). See Fig. 6 for a graphical depiction.

We will argue that the tree representing the best-so-far solution will climb up one f-based level in an expected number of $O(n)$ iterations. The tree can never go down an f-based level, as the fitness of the best-so-far solution never goes down. As there are $\Theta(n)$ f-based levels, this will finish the proof (using a simple waiting time argument).

To show the bound on the time to climb an f-based level, we give bounds on the transition probabilities in the state diagram; in particular, we will prove the claims on the transition probabilities shown in Fig. 6.

Let X be the current (non-optimal) tree. Let $m \leqslant n$ be the number of different nodes in X, and let V be the set of all interior nodes v of X such that

1. all ancestors of v are labeled $\times$;
2. The depth of v is not $D - 1$;
3. v does not evaluate to the maximally possible value, given the depth restriction.

Note that, for any increase of the value of a node from $v \in V$ by a factor of c increases the value of the whole tree by c (if all parts outside the subtree rooted at v stay the same); this uses Item 1 of the above list. Furthermore, $V = \emptyset$ is equivalent to X being the optimal tree.

We consider the following different types of nodes in V. We let $V^{\times}$ be the set of all nodes of V labeled $\times$, V^+ the set of all nodes of V labeled $+$.

We partition the set $V^{\times}$ into the two sets $V^{\times}_{\leqslant 4}$, where one child tree evaluates to $\leqslant 4$, and $V^{\times}_{\geqslant 5}$ of all remaining nodes.

We partition V^+ into the three sets $V^+_{(\leqslant 2, \leqslant 2)}$, $V^+_{(\geqslant 3, 1)}$ and $V^+_{(\geqslant 3, \geqslant 2)}$ as follows. The set $V^+_{(\leqslant 2, \leqslant 2)}$ contains all those nodes that have both child trees evaluate to $\leqslant 2$; $V^+_{(\geqslant 3, 1)}$ contains all those nodes that have one subtree evaluate to at least 3 and the other to 1; finally, the nodes from $V^+_{(\geqslant 3, \geqslant 2)}$ have one child tree evaluating to $\geqslant 3$ and the other to $\geqslant 2$.
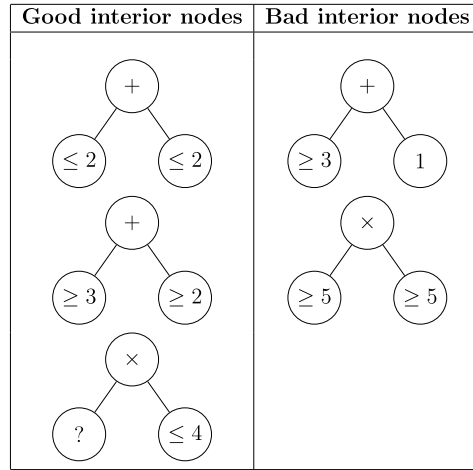
Fig. 7. Good and bad interior nodes.

The different kinds of interior nodes are summarized as follows (some of the types are "good," some are "bad"; this distinction will be used soon). (See Fig. 7.)

For all $v \in V^{\times}_{\leqslant 4} \cup V^{+}_{(\leqslant 2, \leqslant 2)} \cup V^{+}_{(\geqslant 3, \geqslant 2)}$, we call an improvement by a factor of at least 6/5 obtained by changing only elements in the subtree rooted at v an *improvement-move at v*. In this sense, these nodes are "good."

Let such a v be given. We now show that an improvement-move at v has a probability of $\Omega(1/m)$ with the following case analysis.

Case 1. $v \in V^{\times}_{\leqslant 4}$.

Changing the subtree which evaluates to $\leqslant 4$ into a tree that evaluates to one more (which is possible, using Property 3 of nodes in V) and leaving everything else the same (which has probability $\Omega(1/m)$) increases the value of the tree by a factor of at least $5/4 > 6/5$.

Case 2. $v \in V^{+}_{(\leqslant 2, \leqslant 2)}$.

With probability $\Omega(1/m)$, the value of one of the subtrees improves by 1 (this is possible, as all nodes from V do not have optimal value yet); all other elements of X are left unchanged. This will increase the value of v by at least $5/4 \geqslant 6/5$.

Case 3. $v \in V^{+}_{(\geqslant 3, \geqslant 2)}$.

Let $v \in V^{+}_{(\geqslant 3, \geqslant 2)}$. Let a and b be the respective values of the child trees of v. We consider flipping the label of v and leaving all other labels the same (which has a probability of $\Omega(1/m)$). Thus, we get an increase in fitness by a factor of

$$\frac{a \cdot b}{a + b} = \left(\frac{1}{a} + \frac{1}{b}\right)^{-1} \geqslant \left(\frac{1}{2} + \frac{1}{3}\right)^{-1} = \frac{6}{5}.$$

This completes the different cases.

We distinguish the following two kinds of trees. *Bad* trees are trees with $V^{\times}_{\leqslant 4} = V^{+}_{(\leqslant 2, \leqslant 2)} = V^{+}_{(\geqslant 3, \geqslant 2)} = \emptyset$. All other trees are *good*. Intuitively, for good trees, we can find v such that we can make an improvement-move at v. Thus, using the analysis of improvement-moves, we see that all good trees have a chance of increasing in fitness by a factor of at least 6/5 with probability $\Omega(1/m)$.

We proceed now as follows. First, we show that bad trees turn into good ones with probability $\Omega(1/m)$. Then we show that, for good trees, it is at least as likely to improve the fitness by a factor of 6/5 as it is to change into a bad tree (up to constant factors).

Suppose first X is bad; as $V \neq \emptyset$, it is easy to see that there is $v \in V_{(\geqslant 3, 1)}$. The probability for changing the subtree of v with value 1 into a tree with value $\geqslant 2$ and leaving everything else the same is $\Omega(1/m)$. This results in a good tree.

Suppose now X is good; let v be a node closest to the root in X such that $v \in V^{\times}_{\leqslant 4} \cup V^{+}_{(\leqslant 2, \leqslant 2)} \cup V^{+}_{(\geqslant 3, \geqslant 2)}$. Suppose X is turned into a bad tree X'. For this, one of three events has to happen. We let E_1 be the event that, in X', an ancestor of v

is now labeled $+$; we let E_2 be the event that $v \in V_{\geqslant 5}^{\times} \cup V_{(\geqslant 3, 1)}^{+}$ (with respect to X'); we let E_3 be the event that v now evaluates to its maximal value. We let p be the probability that either of E_2 or E_3 happens:

$$p = P(E_2 \cup E_3).$$

We will find events E_2' and E_3' corresponding to the events E_2 and E_3, respectively, such that there is a constant k with $P(E_2') \geqslant k P(E_2)$ and $P(E_3') \geqslant k P(E_3)$; furthermore, we will show $P(E_1) = O(1/m^3)$. Also, in either of the events E_2' and E_3', the f-based level of X' will be strictly higher than in X. This will establish all the transition probabilities given in Fig. 6.

Regarding E_1. An ancestor of v is labeled with $+$ in X'.

Suppose v' is the ancestor of v highest up in the tree that is labeled with $+$ in X'. As X' is bad, one of the children of v' in X' evaluates to 1; however, by choice of v' (high up in the tree), we have that, in X, both children of v' evaluated to at least 5. Thus, in one of the two subtrees rooted at a child of v', *all* interior nodes labeled $+$ had to flip, and there were at least 3 of those (as otherwise the value is at most 4); furthermore, the label of v' changed. This has a probability of $O(1/m^4)$, and there are at most $\log(m)$ possible ancestors of v' where this can happen. Thus, this event has a probability of $O(\log(m)/m^4) \subseteq O(1/m^3)$.

Regarding E_2. v is now a "bad" node in one of the following two ways.

Case 1. $v \in V_{\geqslant 5}^{\times}$ with respect to X'.

For all the three cases for the classification of v with respect to X, we got an increase of fitness by a factor of at least $5/4$ times as much in the (up to constant factors) at least as likely event of not changing anything else to worse values.

Case 2. $v \in V_{(\geqslant 3, 1)}^{+}$ with respect to X'.

Case 2.1. $v \in V_{\leqslant 4}^{\times}$ with respect to X.

In this case, the label of v flipped, which has probability $O(1/m)$. An improvement-move at v has a probability of $\Omega(1/m)$, which finishes this case.

Case 2.2. $v \in V_{(\leqslant 2, \leqslant 2)}^{+}$ with respect to X.

The node v has a value of at least 4 in X'. In order for this not to be improving by a factor of at least $4/3$, both subtrees rooted at the children of v previously had to evaluate to 2. In this case, the change from X to X' required reducing one of the subtrees in value to 1, which has probability $O(1/m)$. An improvement-move at v has a probability of $\Omega(1/m)$, which finishes this case.

Case 2.3. $v \in V_{(\geqslant 3, \geqslant 2)}^{+}$ with respect to X.

In one of the subtrees, all $+$ nodes have been flipped or deleted with probability $O(1/m)$. An improvement-move at v has a probability of $\Omega(1/m)$, which finishes this case.

This finishes the different cases and the analysis for E_2.

Regarding E_3. v evaluates to its maximal value in X'.

Then, if no node outside of the subtree rooted at v flips, we have an improvement-move at v. This is, up to constant factors, as likely as E_3.

This completes the analysis of the different events and shows that, if $O(p + 1/m^3)$ is a bound on the probability for X to become a bad tree, then $\Omega(p + 1/m)$ is a bound on the probability for X' to gain an f-based level over X.

This finishes proving the claims implicit in Fig. 6 and, thus, completes the proof. $\square$

We do not believe that $O(n^2)$ is a tight bound on the runtime for $(1+1)$ GP-multi to optimize MAX-depth-D-$\{+, \times\}$-$\{1\}$. Empirically we were able to obtain the data presented in Fig. 8, which gives the average number of iterations (averaged over 500 independent runs; standard deviation is indicated) for various n. Note that the depth D is indicated on the x-axis, which is roughly $\log n$. The values on the y-axis are divided by $n \log n$, so that a bounded graph corresponds to a runtime of $O(n \log n)$. Thus, it seems that also for $(1+1)$ GP-multi on MAX-depth-D-$\{+, \times\}$-$\{1\}$ the runtime is $O(n \log n)$.
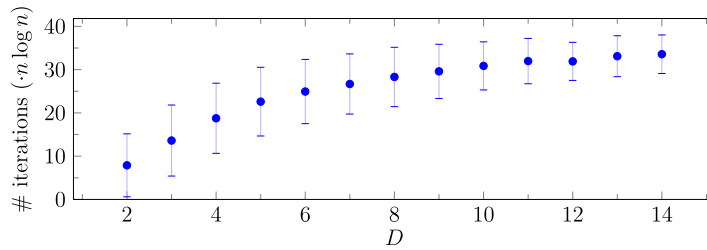
Fig. 8. Runtime for $(1 + 1)$ GP-multi to optimize MAX-depth-D-$\{+, \times\}$-$\{1\}$.

While a lower bound of $\Omega(n \log n)$ follows easily with a coupon collector argument just as in the other settings of this paper, there are strong difficulties for deriving an equivalent upper bound. The $O(n^2)$ bound is based on progress in terms of the value of this tree. We do not believe that such an approach can be used to get a bound of $O(n \log n)$; instead, one would need a potential function based on what nodes are in the tree. The simplest such potential would just be the number of nodes in the tree, but this potential does assign a low value to some trees that are hard to improve (for example, if the left subtree of the root is already complete and the right subtree is only a leaf). Also, it is very hard to control which modifications will be accepted and which will not be, just based on these potentials (this problem disappeared when we used the fitness as the basis for our potential).

5. Biased mutations

In the previous sections, we have analyzed the runtime of different simple GP algorithms for variants of the *Max* problem. One important step was to analyze how the algorithms can make progress by growing the tree using insertion operations. In this section, we will study the growing of trees by insertion operations further. We point out theoretical properties of the insertion operator that is part of the algorithms introduced in Section 2. Using this operator, subtrees which already have many nodes are more likely to be expanded. Depending on the problem at hand, a more equal growth might be preferable. The use of biased mutation operators for variable-length representations has recently been explored in [2]. We present an insertion operator leading to a more balanced tree growth and show improved upper bounds when using this operator.

5.1. Growing trees by biased insertions

We consider the stochastic process detailed in Algorithm 2 below for growing an infinite binary tree. We study the expected time until a particular node v in the infinite binary tree is created. The following theorem shows that the expected time to create any node with depth > 1 is infinite.

Algorithm 2: Standard insertions.
---
1 Start with a tree X consisting of a single leaf r;
2 Iteratively replace a leaf in X chosen uniformly at random by a node with two leaves as children;

Theorem 11. *For any given node v with depth strictly greater than 1 in the complete infinite binary tree, the expected number of iterations until v is created when growing the tree with standard insertions is infinite.*

Proof. Fix a child v of the root. We compute the expected time until v is expanded as follows. The earliest iteration that v can be expanded is iteration 2, with probability $1/2$; in iteration 3, if v is not expanded already (with probability $1/2$), v is expanded with probability $1/3$, and so on. This gives the following formula for the expected time until v is expanded:

$$\sum_{i=2}^{\infty} i \cdot \frac{1}{i} \prod_{j=1}^{i-2} \frac{j}{j+1} = \sum_{i=2}^{\infty} \frac{1}{i-1} = \infty.$$

As no node can be created before any of its ancestors, this finishes the proof. □

With an alternate model for growing trees, we get a more balanced result as follows. Consider the stochastic process for growing an infinite binary tree outlined in Algorithm 3 below. It is easy to see that, in a binary tree, Algorithm 3 gives a probability distribution on the leaves.

Algorithm 3: Biased insertions.
---
1 Start with a tree X consisting of a single leaf r;
2 Iteratively replace a randomly chosen leaf in X, where a leaf of distance d to the root has probability 2^{-d} of being chosen, by a node with two leaves as children;

Theorem 12. *Using Algorithm 3, the expected number of iterations until a particular node v is created is $\Theta(2^d)$, where d is the distance of v to the root.*

Proof. Let v be any node in the infinite binary tree, let d be its distance to the root. Using induction on d with trivial base case, we assume the expected time until the parent of v is expanded to be $2^d - 1$. Once the parent of v is expanded, the expected time until v is expanded is 2^d iterations. This results in a total number of iterations of $2^d - 1 + 2^d = 2^{d+1} - 1$. □

5.2. Runtime analysis for biased mutation

Inspired by these remarks, we modify `insert` and `delete` so that a leaf is not chosen uniformly, but instead with probability 2^{-d}, where d is the distance of the leaf to the root. We call the versions of $(1 + 1)$ GP based on these modified variation operators $(1 + 1)$ GP-balanced-single and $(1 + 1)$ GP-balanced-multi.

Theorem 13. *For any $t > 0$, the expected time for $(1 + 1)$ GP-balanced-single to optimize MAX-depth-D-{+}-{t} is $O(n \log n)$, asymptotically independent of t.*

Proof. We argue similarly as in Theorem 4. The optimal solution is the complete binary tree of depth D and deletions are never accepted.

Let X be a non-optimal tree; suppose all nodes on some level $k - 1$ are interior nodes (counting levels as distance to the root). Using a simple coupon collector argument, all nodes on level k are made interior nodes within an expected number of iterations of at most $\sum_{i=1}^{2^k} \frac{2^k}{i} = O(2^k k)$. Thus, summing over all levels, we get a total expected running time for $(1 + 1)$ GP-balanced-single of $\sum_{k=1}^{\log(n)} O(2^k k) = O(n \log n)$. This last step is entailed by a well-known summation formula, or by observing that the terms grow superexponentially, and, hence, the last term dominates. □

Theorem 14. *For any $t > 0$, the expected time for $(1 + 1)$ GP-balanced-single to optimize MAX-depth-D-{+, ×}-{t} is $O(n \log n)$, asymptotically independent of t.*

Proof. By the same arguments used in the proofs of Theorems 5, 6 and 7, for any $t \geqslant 0$, an unfixed position with a fixed parent becomes fixed after a substitution or an insert operation involving a function node labeled $+$.

Let X be a non-optimal tree. Suppose X is fixed to level $k - 1$ (in the sense of Definition 2) for some $k > 0$. If there are i positions on level k that are unfixed, under biased mutation, the probability that the correct operation fixes any one of them is $(i/3) \cdot 2^{-k}$. In an analogous way to the proof of Theorem 13, the claim follows by applying the coupon collector argument for collecting the remaining fixed positions in level k, and finally by taking the sum over all levels. □

Furthermore, for $(1 + 1)$ GP-balanced-multi we can also obtain much improved results.

Theorem 15. *For any $t > 0$, the expected time for $(1 + 1)$ GP-balanced-multi to optimize MAX-depth-D-{+}-{t} is $O(n \log n)$.*

Proof. We only consider single insertion operations that happen in each step with probability $1/(3e)$ and take a fitness-based partitions approach with respect to the number of nodes m in the tree. Note, that the number of nodes in the tree cannot decrease during the optimization process.

Let X be a non-optimal tree with m nodes; thus, the tree has at most $m/2$ leaves at maximum depth D. The probability to select one particular leaf at a maximal depth D in an insertion step is $2^{-D} = 2^{-(\log(n+1)-1)} = 2/(n + 1)$. Expanding one of the leaves at maximal depth has thus a probability of at most $\frac{m}{2} \cdot \frac{2}{n+1} = m/(n + 1)$; hence, expanding one of the leaves which is *not* at maximal depth has a probability of at least $1 - m/(n + 1)$. Expanding a leaf at non-maximal depth and changing nothing else will give a strictly larger tree (recall that making only a single change has constant probability).
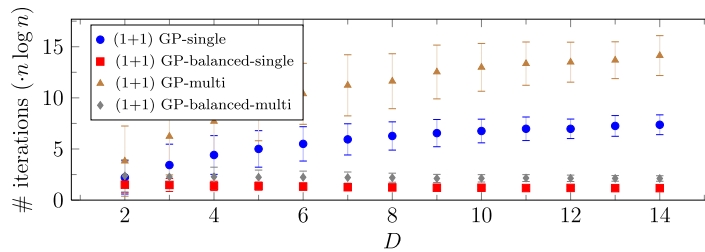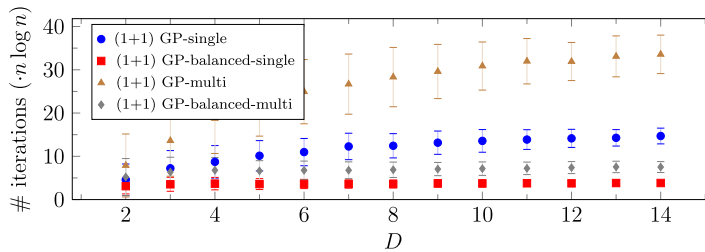
Thus, if there are still $k = n - m$ nodes missing for the complete binary tree of depth $\log(n + 1)$, we have a probability of at least

$$\frac{1}{3e} \cdot \left(1 - m/(n + 1)\right) = \frac{1}{3e} \cdot \left(1 - (n - k)/(n + 1)\right) = \frac{1}{3e} \cdot (k + 1)/(n + 1) = \Omega(k/n)$$

to decrease the number of missing nodes in X by 1 in the next mutation step. Summing up the waiting times for the different values of k, we get

$$\sum_{k=1}^{n} O(n/k) = O(n \log n),$$

which completes the proof. □

Fig. 9. Runtimes on MAX-depth-D-{+}-{1}.

Fig. 10. Runtimes on MAX-depth-D-{+, ×}-{1}.

5.3. Experimental investigations

The goal of the experimental section is to evaluate the biased mutation operator, which implies a more balanced growth process of the syntax tree. The theoretical bounds are also $O(n \log n)$ when using single (non-balanced) operators, but Theorems 11 and 12 suggest that there might be an advantage when working with the balanced mutation operator.

We evaluated the different algorithms, $(1+1)$ GP-single and $(1+1)$ GP-multi with and without balanced tree growth, on MAX-depth-D-{+}-{1} for D from 2 to 14. Fig. 9 gives the results indicating the mean, averaged over 500 independent runs, and the standard deviation. On the x-axis is the value of D, the y-axis shows the number of iterations, divided by $n \log n$. Thus, any bounded graph corresponds to a runtime of $O(n \log n)$. Similarly, Fig. 10 gives results on MAX-depth-D-{+, ×}-{1}.

The figures show that single operations are usually outperforming multi operations when considering the same mutation approach. Furthermore, it can be observed that using the balanced mutation operator yields significant speed-ups by factors of about 5 to 6 for the both the single and the multi operations compared to the standard approach. We also see a reduction in variance for the runs that used a balanced mutation operator.

6. Discussion and conclusions

Analyzing the computational complexity of genetic programming algorithms on exemplary problems can help to understand the behavior of these algorithms in a rigorous manner. The runtime bounds presented in this paper show that simple mutation provably helps to solve variants of the *Max* problem. For the mutation operators that use a single atomic operation in each mutation step, we have shown that an optimal solution is obtained for all positive values of t in expected time $O(n \log n)$. This asymptotic runtime result holds for any fixed positive value at the leaves.

Furthermore, we have studied mutation in greater detail and shown that biased mutation operators have nicer theoretical properties with respect to the process of growing trees. Our experimental results show that the biased mutation operator designed from these insights has a better runtime behavior than the unbiased mutation operator we studied.

The simplicity of the $(1+1)$ GP serves our purposes of proving runtime bounds, but its efficiency on the *Max* problem highlights a few general issues. First, it motivates a reminder that mutation is likely neglected more often than it should be in the genetic programming community. HVL-Prime is attractive because it is an incremental operator. While it takes small steps, if these steps can also be combined, like in the case of $(1+1)$ GP-multi, it offers a different means of variation. Since tree growth and fitness improvement are coupled but not controlled nor well-understood, diverse variation operators may be helpful. While mutation works well specifically on the *Max* problem, this success may not be general to mutation. In practice, $(1+1)$ GP algorithms could be useful for scaling a problem to a point where it no longer can be easily solved by hill climbing. At that point, conventional genetic programming operators may be more effective.

References

[1] A. Auger, B. Doerr (Eds.), Theory of Randomized Search Heuristics: Foundations and Recent Developments, World Scientific, 2011.
[2] S. Cathabard, P.K. Lehre, X. Yao, Non-uniform mutation rates for problems with unknown solution lengths, in: H.-G. Beyer, W.B. Langdon (Eds.), FOGA, ACM, 2011, pp. 173–180.
[3] G. Durrett, F. Neumann, U.-M. O'Reilly, Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics, in: Proc. of FOGA'11, 2011, pp. 69–80.
[4] V. Feldman, Evolvability from learning algorithms, in: C. Dwork (Ed.), STOC, ACM, 2008, pp. 619–628.
[5] C. Gathercole, P. Ross, An adverse interaction between crossover and restricted tree depth in genetic programming, in: Proceedings of the First Annual Conference on Genetic Programming, GECCO'96, 1996, pp. 291–296.
[6] D.E. Goldberg, U.-M. O'Reilly, Where does the good stuff go, and why? How contextual semantics influences program structure in simple genetic programming, in: W. Banzhaf, R. Poli, M. Schoenauer, T.C. Fogarty (Eds.), EuroGP, in: Lecture Notes in Comput. Sci., vol. 1391, Springer, 1998, pp. 16–36.
[7] G. Ivchenko, Limit theorems in an occupancy problem, Theory Probab. Appl. 16 (2) (1971) 293–307.
[8] T. Kötzing, F. Neumann, R. Spöhel, PAC learning and genetic programming, in: N. Krasnogor, P.L. Lanzi (Eds.), GECCO, ACM, 2011, pp. 2091–2096.
[9] T. Kötzing, A.M. Sutton, F. Neumann, U.-M. O'Reilly, The max problem revisited: the importance of mutation in genetic programming, in: T. Soule, J.H. Moore (Eds.), Genetic and Evolutionary Computation Conference, GECCO'12, Philadelphia, PA, USA, July 7–11, 2012, ACM, 2012, pp. 1333–1340.
[10] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, USA, 1992.
[11] W.B. Langdon, R. Poli, An analysis of the MAX problem in genetic programming, in: Advances in Genetic Programming 3, MIT Press, 1997, pp. 301–323 (Chapter 13).
[12] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.
[13] F. Neumann, Computational complexity analysis of multi-objective genetic programming, in: T. Soule, J.H. Moore (Eds.), Genetic and Evolutionary Computation Conference, GECCO'12, Philadelphia, PA, USA, July 7–11, 2012, ACM, 2012, pp. 799–806.
[14] F. Neumann, C. Witt, Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity, Springer, 2010.
[15] U.-M. O'Reilly, An analysis of genetic programming, PhD thesis, Carleton University, Ottawa–Carleton Institute for Computer Science, Ottawa, Ontario, Canada, 22 Sept. 1995.
[16] U.-M. O'Reilly, F. Oppacher, Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing, in: Y. Davidor, H.-P. Schwefel, R. Manner (Eds.), Parallel Problem Solving from Nature – PPSN III, in: Lecture Notes in Comput. Sci., vol. 866, Springer-Verlag, 1994, pp. 397–406.
[17] L.G. Valiant, Evolvability, J. ACM 56 (1) (2009).