The Max Problem Revisited: The Importance of Mutation in Genetic Programming

Timo Kötzing
Algorithms and Complexity
Max-Planck-Institut für Informatik
66123 Saarbrücken, Germany

Andrew M. Sutton, Frank Neumann
School of Computer Science
University of Adelaide
Adelaide, SA 5005, Australia

Una-May O'Reilly
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

ABSTRACT

This paper contributes to the rigorous understanding of genetic programming algorithms by providing runtime complexity analyses of the well-studied *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 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.

Categories and Subject Descriptors

F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity

General Terms

Theory, Algorithms, Performance

Keywords

Genetic Programming, Mutation, Theory, Runtime Analysis

1. INTRODUCTION

The goal of this paper is to advance the computational complexity analysis of genetic programming. This type of analysis has significantly increased the theoretical understanding of other types of evolutionary algorithms (see the books [1, 12] for a comprehensive presentation).

Genetic programming (GP) refers to a class of evolutionary algorithms that evolve, for a specific task, executable structures, such as computer programs. Pioneered by Koza [8] in the early 1990's, genetic programming has been demonstrably successful at solving human competitive programming problems arising from diverse domains. There are numerous examples of genetic programming, where subtree crossover is the only variation operator used. The *Max* problem [5] was 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, leaving a mutation operator as the only common, but ineffective, route to discovery of fitter trees." (abstract, [5]) 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" (pg 229, [9]).

We revisit the *Max* problem also because its solution space is easy to think about and its solutions are known in advance. It is parameterized in a manner that changes its solution and its parameterizations allow incremental difficulty. Our algorithms employ the HVL-Prime operator introduced in [3]. HVL-Prime 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 deleting the tree by the minimum number of nodes possible. It satisfies the need to search trees which 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 [11] to parsimony and multi-objective GP algorithms for generalizations of these two problems. Furthermore, GP algorithms have been studied in the PAC learning framework [7] and general studies on the learnability of evolutionary algorithms for Boolean functions have already been carried out before in [4, 15]. ORDER and MAJORITY are in some sense easy to optimize as they have independent problem semantics. The (1+1) GP al-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'12, July 7–11, 2012, Philadelphia, Pennsylvania, USA.
Copyright 2012 ACM 978-1-4503-1177-9/12/07 ...$10.00.

gorithms studied in [3] use a variable-length representation that is important since syntax trees can usually grow and shrink during the optimization process. 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. The *Max* problem has the important property that different functions such as addition and multiplication are available but the set of terminals consists of the simplest forms, i.e., only of one specific constant. We will analyze variants of (1+1) GP on different variants of the *Max* problem. The problems distinguish each other in the set of functions and terminals available for evolution of maximum solutions.

On this occasion, rather than use *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) [13, 14] is one such example. SIHC is most similar to our (1+1) mutation-based hillclimber. 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 $O(n \log^2 n)$ when one mutation operation precedes each fitness evaluation. When multiple mutations are successively applied before each fitness evaluation, the time complexity bound is $O(n^2)$. This bound can be reduced to $O(n \log n)$ if the mutations are biased to replace a random leaf with distance d from the root with probability 2^{-d} . It seems surprising that the *Max* problem is so solved with a hill climbing algorithm that uses mutation when, in a population based setting, neither subtree crossover alone or in tandem with mutation is effective. 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 would suggest that there is low cost and possible value to evaluating the scale at which a mutation-based hill climber effectively solves one's 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.

We proceed as follows. In Section 2, we formally introduce the problem and algorithms that are subject to our investigations. 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 more general (1+1) GP-multi in Section 4. Finally, we show in Section 5 how biased mutation operators can lead to better runtime bounds. We finish with some discussion and conclusions.

2. THE MAX PROBLEM

The task for the *Max* problem is to find a program (as given by a syntax tree) which 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 denote by $n = 2^{D+1} - 1$ the maximal number of nodes in a binary tree of depth D.

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 [9] 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 Problems.

- The problem MAX-depth-D-$\{+\}$-$\{1\}$ is the perhaps simplest Max Problem; the optimal tree is a full binary tree, with $+$ at interior nodes and 1s at leaves. The value of an optimal solution is 2^D.

- MAX-depth-D-$\{+, \times\}$-$\{t\}$ 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; optimizing nodes first suggests to label them $+$ (while they are parents of leaves) and then $\times$ (when both child trees evaluate to something ≥ 2).
  - If $t = 0.5$, then there are three layers of $+$ (counting from just above the leaves) before $\times$ is strictly preferred.

For a current tree X, we will denote by $F(X)$ 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.

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. If `insert` results in a tree that has depth greater than D, it is not accepted. In this case, `insert` behaves like a null operation.

- 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 which 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. We will implicitly use the observation that k following this distribution is, with constant probability, equal to 1; similarly, it is with constant probability equal to 2.

Algorithm 1: (1+1) GP

```
1 Choose X as a leaf from T u.a.r.;
2 while optimum not reached do
3     X' ← X;
4     Choose k;
5     for i = 1 to k times do
6         Choose m ∈ {substitute, insert, delete}
          u.a.r.;
7         X' ← m(X');
8     if f(X') ≥ f(X) then X ← 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 $O(n \log^2 n)$. We first introduce some definitions that facilitate the analysis.

Definition 1. *We say a position in a tree X is* fixed *if it contains a node that cannot be deleted without reducing the fitness.*

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

We now show that, as long as the probability to fix a node is not too small, the (1+1) GP-single can efficiently fix all nodes 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 tree is fixed to level m is $O(nm^2)$.*

Proof. In a nonempty tree, the root position is already fixed. Hence, without loss of generality, we start with a tree that is already fixed to some level $k \geq 0$.

We first bound the expected time to transform X into a tree X' that is fixed to level $k+1$. Let i be the number of unfixed positions in level $k+1$. By the supposition of the claim, the probability of reducing the number of unfixed nodes at this level is at least $i/(cn)$ for a constant c.

Since the number of unfixed nodes at level $k+1$ will never decrease during a run of the algorithm, the time to transform X into a tree X' that is fixed to level $k+1$ is bounded by

$$\sum_{i=1}^{2^{k+1}} \frac{cn}{i} = O(n \log 2^{k+1}) = O(nk).$$

Summing up the runtime for distinct values of k we get an upper bound on the time to fix an arbitrary tree to level m.

$$\sum_{k=0}^{m} O(nk) = O(nm^2).$$

□

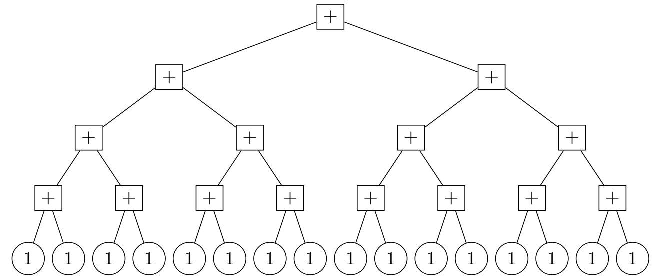

Figure 1: An optimal tree for MAX-depth-D-$\{+\}$-$\{1\}$ with $D = 4$.

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-$\{+\}$-$\{1\}$. See Figure 1 for an illustration of the optimal tree for $D = 4$ on this variant.

Theorem 4. *The expected time for the (1+1) GP-single to optimize MAX-depth-D-$\{+\}$-$\{1\}$ is in $O(n \log^2 n)$.*

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 $O(nD^2) = O(n \log^2 n)$ since $D = \log n$. □

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 Figure 2 for the optimal depth-four tree when $t = 2$.

Theorem 5. *The expected time for the (1+1) GP-single to optimize MAX-depth-D-$\{+, \times\}$-$\{t\}$, for all $t > 1$, is in $O(n \log^2 n)$.*

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 $O(n \log^2 n)$ 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 $z \geq t$, such a transformation always results in an improving fitness and by the coupon collector theorem [10], 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 $z \geq 2t$, so each specific substitution guarantees an improving move. Again by the coupon

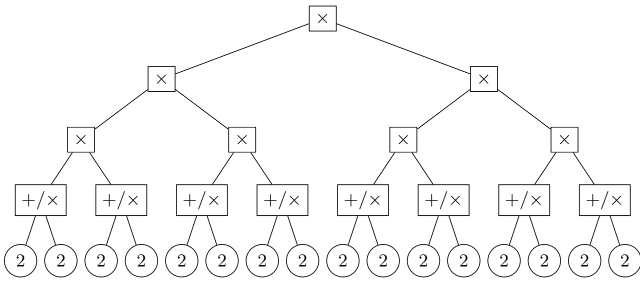

Figure 2: An optimal tree for MAX-depth-D-$\{+,\times\}$-$\{2\}$ with $D=4$.

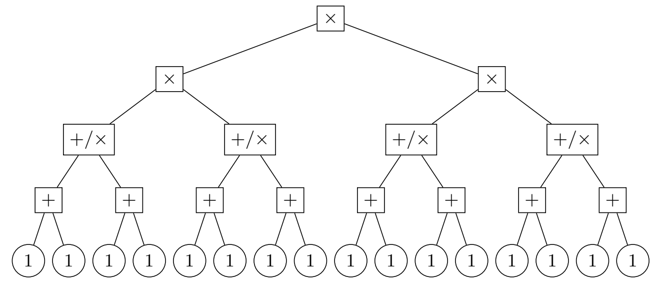

Figure 3: An optimal tree for MAX-depth-D-$\{+,\times\}$-$\{1\}$ with $D=4$.

collector theorem, the correct substitutions have been made after $O(n\log n)$ operations in expectation. In all cases, the expected time to transform the function nodes is dominated by the time to fix the tree to level D. □

We now consider the case when the terminal values are constrained to 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 $O(n\log^2 n)$.*

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 × nodes, and all remaining levels consist only of × 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 × 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 × nodes (see Figure 3).

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 $O(n\log^2 n)$.

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. Thus the expected time to solve the tree is dominated by the the time to fix the tree to level D. □

3.1 Terminals of smaller value

Gathercole and Ross [5] showed that the MAX problem can be made progressively more difficult for traditional GP by decreasing the size of the constant terminal. For example, in their paper they used the values $t=0.5$ and $t=0.25$. We find in the case of the (1+1) GP-single that such a decrease does not affect the asymptotic character of the runtime.

THEOREM 7. *The expected time for the (1+1) GP-single to optimize MAX-depth-D-$\{+,\times\}$-$\{0.5\}$ is in $O(n\log^2 n)$.*

Figure 4 illustrates the depth-four optimal tree. Before proving Theorem 7, we make a few observations that facilitate the proof. Let us first define a "light node" as any interior node labeled × with at least one child evaluating to strictly less than 1. We make the following remark.

REMARK 8. *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. Obviously a deletion cannot create a light node. Furthermore, an insertion cannot create a light node since the value at the root of the new tree would be half the value at the root of the old tree, reducing the overall fitness of the tree. Finally, we show that a substitution cannot create a light node. Suppose v is a node labeled + whose children evaluate to x and y. The substitution operation that relabels v with × is only accepted if $xy\geq x+y$. Without loss of generality, suppose $x\leq y$. Since $x\geq 1+y/x$, it follows that the resulting node cannot be light.

Now suppose we start with an initial tree that contains k light nodes. Let v be a particular light node whose children evaluate to x and y where $x<1$. A substitution at v is always accepted since $xy<y<y+x$. 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. □

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

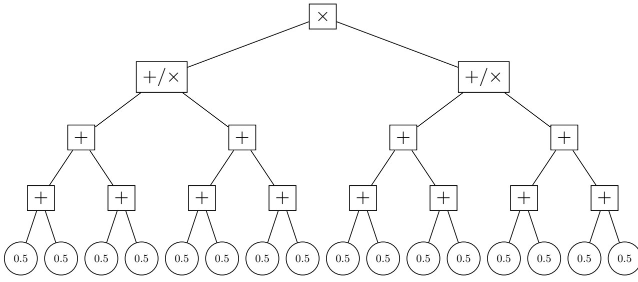

Figure 4: An optimal tree for MAX-depth-D-{$+, \times$}-{0.5} with $D = 4$.

matter is that a deletion can only occur when the parent of the deleted leaf is labeled $\times$. Such a node is light by definition, hence if there are no light nodes, there are no deletions possible.

PROOF OF THEOREM 7. 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 $O(n \log^2 n)$ expected iterations. After fixation, the interior node labels are corrected after $O(n \log n)$ expected substitutions. □

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

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

□

In this section we will state and prove a theorem about the performance of (1+1) GP-multi on MAX-depth-D-{$+, \times$}-{1} (see Theorem 10). 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 application of the HVL-Prime operator 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

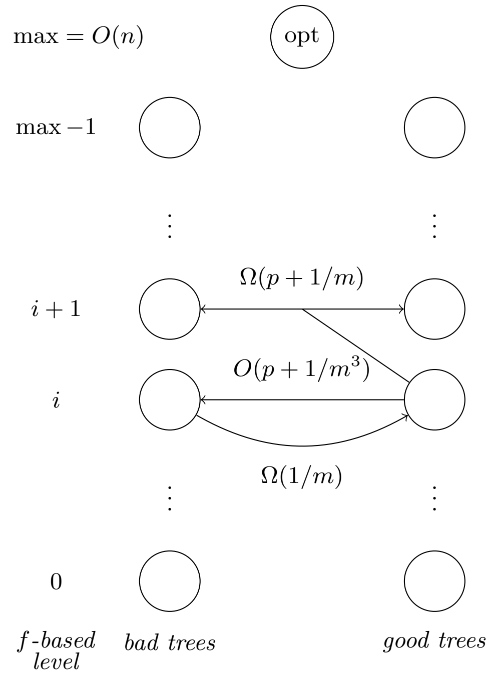

Figure 5: State diagram and transition probabilities between different types (good/bad) of tree and different f-based levels dependent on the number m of nodes in the tree.

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 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 Figure 5 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 Figure 5.

Let X be the current (non-optimal) tree. We let $m \leq n$ be the number of different nodes in X. We 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 $V^{\times}_{\leq 4}$, where one child tree evaluates to ≤ 4, and $V^{\times}_{\geq 5}$ of all remaining nodes.

We partition V^{+} into the three sets $V^{+}_{(\leq 2, \leq 2)}$, $V^{+}_{(\geq 3,1)}$ and $V^{+}_{(\geq 3, \geq 2)}$ as follows. The set $V^{+}_{(\leq 2, \leq 2)}$ contains all those nodes that have both child tree evaluate to ≤ 2; $V^{+}_{(\geq 3,1)}$ contains all those nodes that have one subtree evaluate to at least 3 and the other to 1; finally, the nodes from $V^{+}_{(\geq 3, \geq 2)}$ have one child tree evaluating to ≥ 3 and the other to ≥ 2.

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).

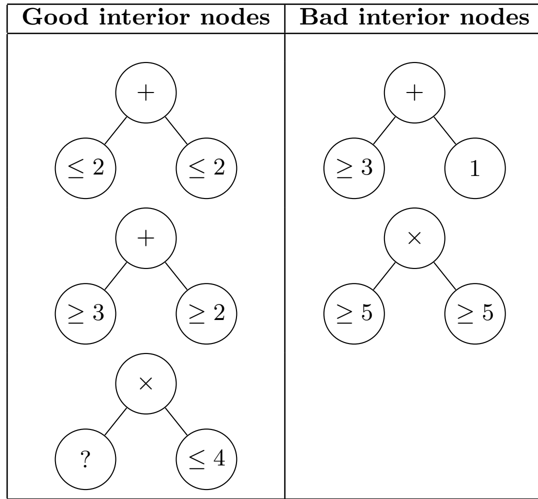

Figure 6: Good and bad interior nodes

For all $v \in V^{\times}_{\leq 4} \cup V^{+}_{(\leq 2, \leq 2)} \cup V^{+}_{(\geq 3, \geq 2)}$, 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: $v \in V^{\times}_{\leq 4}$.
Changing the subtree which evaluates to ≤ 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: $v \in V^{+}_{(\leq 2, \leq 2)}$.
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 $5/4 \geq 6/5$.

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

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

This completes the different cases.

We distinguish the following two kinds of trees. *Bad* trees are trees with $V^{\times}_{\leq 4} = V^{+}_{(\leq 2, \leq 2)} = V^{+}_{(\geq 3, \geq 2)} = \emptyset$. 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 $v \in V_{(\geq 3,1)}$. The probability for changing the subtree of v with value 1 into a tree with value ≥ 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 $v \in V^{\times}_{\leq 4} \cup V^{+}_{(\leq 2, \leq 2)} \cup V^{+}_{(\geq 3, \geq 2)}$. 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 $v \in V^{\times}_{\geq 5} \cup V^{+}_{(\geq 3,1)}$ (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 $P(E_2') \geq kP(E_2)$ and $P(E_3') \geq kP(E_3)$; 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 Figure 5.

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: $v \in V^{\times}_{\geq 5}$ 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: $v \in V^{+}_{(\geq 3,1)}$ with respect to X'. *Case 2.1:* $v \in V^{\times}_{\leq 4}$ 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:* $v \in V^{+}_{(\leq 2,\leq 2)}$ 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:* $v \in V^{+}_{(\geq 3,\geq 2)}$ 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 Figure 5 and, thus, completes the proof. □

5. BIASED MUTATIONS

In the previous sections, we have analyzed the runtime of different simple GP algorithms for variants of the 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 following stochastic process growing an infinite binary tree.

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;

It is easy to see that, in a binary tree, this gives a probability distribution on the leaves.

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. *The expected time for (1+1) GP-balanced-single to optimize MAX-depth-D-{+}-{1} is $O(n \log n)$.*

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 super-exponentially, and, hence, the last term dominates. □

Table 1: Runtime bounds for the *Max* problem.

Problem	(1+1) GP-		
MAX-depth-D-	single	multi	balanced
{+}-{1}	$O(n \log^2 n)$	$O(n^2)$	$O(n \log n)$
{+×}-{t}, t>1	$O(n \log^2 n)$		
{+×}-{t}, t=1	$O(n \log^2 n)$	$O(n^2)$	$O(n \log n)$
{+×}-{t}, t=1/2	$O(n \log^2 n)$		

Combining the above analysis with the arguments proving Theorem 6, we have the following result.

THEOREM 14. *The expected time for (1+1) GP-balanced-single to optimize MAX-depth-D-*$\{+, \times\}$-$\{1\}$ *is* $O(n \log n)$.

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

THEOREM 15. *The expected time for (1+1) GP-balanced-multi to optimize MAX-depth-D-*$\{+\}$-$\{1\}$ *is* $O(n \log n)$.

PROOF. Let a non-optimal tree with m nodes; thus, the tree has at most $m/2$ leaves at maximal depth. Expanding one of the leaves at maximal depth has thus a probability of at most $m/(n+1)$; hence, expanding one of the leaves which is *not* at maximal depth has a probability of $1 - m/(n+1)$. 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 $\Omega(k/n)$ to decrease the number of missing nodes by 1. Furthermore, the number of missing nodes never increases. Using a multiplicative drift theorem, the result follows. □

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 (summarized in Table 1) show that simple mutation provably helps to solve the variants of the *Max* problem presented in this paper. Furthermore, we have studied mutation in greater detail and shown that biased mutation operators lead to improved runtime bounds of $O(n \log n)$ which can be considered optimal as such algorithms usually encounter the coupon collector effect.

One might remark how a simple algorithm like (1+1) GP is more effective than conventional genetic programming with its population and tree-based crossover on the *Max* problem and consider what this suggests more generally. First, it motivates a reminder that mutation is likely neglected more often than it should be. 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. However, maintaining simple (1+1) GP algorithms in one's library might allow for scaling a problem to a point where it no longer can be easily solved by hill climbing. At that point, conventional genetic programming may be merited.

7. REFERENCES

[1] A. Auger and B. Doerr, editors. *Theory of Randomized Search Heuristics: Foundations and Recent Developments.* World Scientific, 2011.

[2] S. Cathabard, P. K. Lehre, and X. Yao. Non-uniform mutation rates for problems with unknown solution lengths. In H.-G. Beyer and W. B. Langdon, editors, *FOGA*, pages 173–180. ACM, 2011.

[3] G. Durrett, F. Neumann, and U.-M. O'Reilly. Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics. In *Proc. of FOGA'11*, pages 69–80, 2011.

[4] V. Feldman. Evolvability from learning algorithms. In C. Dwork, editor, *STOC*, pages 619–628. ACM, 2008.

[5] C. Gathercole and 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, pages 291–296, 1996.

[6] D. E. Goldberg and 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, and T. C. Fogarty, editors, *EuroGP*, volume 1391 of *Lecture Notes in Computer Science*, pages 16–36. Springer, 1998.

[7] T. Kötzing, F. Neumann, and R. Spöhel. PAC learning and genetic programming. In N. Krasnogor and P. L. Lanzi, editors, *GECCO*, pages 2091–2096. ACM, 2011.

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

[9] W. B. Langdon and R. Poli. An analysis of the MAX problem in genetic programming. In *Advances in Genetic Programming 3, chapter 13*, pages 301–323. MIT Press, 1997.

[10] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[11] F. Neumann. Computational complexity analysis of multi-objective genetic programming. In *GECCO*. ACM, 2012. to appear.

[12] F. Neumann and C. Witt. *Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity.* Springer, 2010.

[13] 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.

[14] U.-M. O'Reilly and F. Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Y. Davidor, H.-P. Schwefel, and R. Manner, editors, *Parallel Problem Solving from Nature – PPSN III*, number 866 in Lecture Notes in Computer Science, pages 397–406. Springer-Verlag, 1994.

[15] L. G. Valiant. Evolvability. *J. ACM*, 56(1), 2009.