Theoretical Computer Science 545 (2014) 94–107

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

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

Keywords:
Genetic programming
Mutation
Theory
Runtime analysis

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.

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).

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2013.06.014

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.