On the Performance of Different Genetic Programming Approaches for the SORTING Problem

Markus Wagner markus.wagner@adelaide.edu.au
Optimisation and Logistics, University of Adelaide, Adelaide, Australia

Frank Neumann frank.neumann@adelaide.edu.au
Optimisation and Logistics, University of Adelaide, Adelaide, Australia

Tommaso Urli tommaso.urli@nicta.com.au
DIEGM, Università degli Studi di Udine, Udine, Italy

doi:10.1162/EVCO_a_00149

Abstract

In genetic programming, the size of a solution is typically not specified in advance, and solutions of larger size may have a larger benefit. The flexibility often comes at the cost of the so-called bloat problem: individuals grow without providing additional benefit to the quality of solutions, and the additional elements can block the optimization process. Consequently, problems that are relatively easy to optimize cannot be handled by variable-length evolutionary algorithms. In this article, we analyze different single- and multiobjective algorithms on the sorting problem, a problem that typically lacks independent and additive fitness structures. We complement the theoretical results with comprehensive experiments to indicate the tightness of existing bounds, and to indicate bounds where theoretical results are missing.

Keywords

Computational complexity, genetic programming, variable-length representation, sortedness, single-objective optimization, multiobjective optimization.

1 Introduction

Evolutionary algorithms using variable-length representations have been applied in different problem domains (Falco et al., 2005; Lee and Antonsson, 2000). They are, in particular, useful if there is a trade-off between the quality of a solution and its complexity in terms of the size of its representation. This happens frequently in the area of regression where a more complex model can give a better fit to the given data.

Genetic programming (GP) (Koza, 1992) is the most prominent example of a variable-length evolutionary algorithm, as it often evolves tree-like solutions for a given problem. Its main application area lies in the field of symbolic regression. The first computational complexity results on this type of algorithm were obtained following the line of successful research on evolutionary algorithms with fixed-length representation (Auger and Doerr, 2011; Neumann and Witt, 2010, for an overview). In general, variable-length representations increase the search space significantly, and it is desirable to better understand the behavior of algorithms using such representations from a theoretical point of view.

Manuscript received: October 31, 2014; revised: March 4, 2015; accepted: April 2, 2015.

For example, Cathabard et al. (2011) investigated nonuniform mutation rates for problems with unknown solution lengths. A simple evolutionary algorithm was used to find a bit string with an unknown number of leading 1s, and although the bit string had some predetermined maximum length, only an unknown number of initial bits was used by the fitness function. A simple tree-based genetic programming algorithm was investigated by Durrett et al. (2011). The problems were separable, with independent and additive fitness structures. Similarly, Kötzing et al. (2012) analyzed simple GP algorithms for the MAX problem. A new form of GP called geometric semantic genetic programming was investigated, with positive results for Boolean, classification, and regression domains (see, e.g., Moraglio et al., 2013; Mambrini et al., 2013).

Many evolutionary algorithms that work with a variable-length representation do not work (in their most basic variant) with a form of bloat control. One popular way to deal with the bloat problem is inspired by Occam's Razor: in case two solutions are equal in quality, the solution of lower complexity is preferred. Another frequently taken approach to coping with the bloat problem is the use of multicriteria approaches that use sets of solutions representing the different trade-offs according to the original goal function and the complexity of a solution. Such multicriteria approaches are used even in industrial GP packages such as Datamodeler by Evolved Analytics LLC (2010). Both approaches to coping with the bloat problem have been examined for different problems in the context of genetic programming (Neumann, 2012; Wagner and Neumann, 2012; Nguyen et al., 2013).

In this article, we investigate the sorting problem, which is one of the fundamental problems in computer science.[1] In addition, the sorting problem is the first combinatorial optimization problem for which computational complexity results have been obtained in the area of discrete evolutionary algorithms (Scharnow et al., 2004; Doerr and Happ, 2008). Scharnow et al. (2004) formulated sorting as a problem where the task is the minimization of unsortedness (or the maximization of sortedness) of a given permutation of the input elements. Several different functions have been explored in the past to measure unsortedness, and they have been studied with respect to the difficulty of being optimized by permutation-based evolutionary algorithms. Depending on the chosen measure and in contrast to the problems WORDER and WMAJORITY (see, e.g., Nguyen et al., 2013), the sorting problem cannot typically be split into subproblems that can be solved independently. Consequently, the dependencies between the subproblems can significantly impact the time needed to solve the overall optimization problem.

With our work, we continue the analyses started by Wagner and Neumann (2012), which focussed on the advantages of a parsimonious algorithm over a multiobjective one. Here, we present several analyses for a total of three single- and multiobjective algorithms using five sortedness measurements. Despite our analyses, no (or no exact) runtime bounds are given for different combinations of algorithms and problems. Because of this, we explore experimentally the open cases and questions. The intention is that this will guide further rigorous analyses (similar to Lässig and Sudholt, 2010; Briest et al., 2004; Urli et al., 2012) by exploring the important measures within a computational complexity analysis of the algorithms. We complement the theoretical results with conjectures about the expected optimization times for the variants lacking a formal proof. In our experimental investigations, we concentrate on important measures, such as the size of the largest tree during the run of the single-objective algorithms

[1]Short versions of Sections 2– 6 of this article were published in Wagner and Neumann (2014).

analyzed by Durrett et al. (2011) and the maximum population size of the multiobjective algorithm analyzed by Neumann (2012). In both articles, these measures have different implications for the computational complexities of the analyzed algorithms.

This article is organized as follows. We first introduce the sorting problem in Section 2. Then we present the different genetic programming algorithms, which are analyzed in Section 3. In Section 4 we study the single-objective approach, in Section 5 the parsimony approach, and in Section 6 the multiobjective approach. The experimental investigations on the behaviors of (1+1)-GP and SMO-GP follow in Section 7. We finish with some conjectures and concluding remarks in Section 8.

2 Preliminaries

Our goal is to investigate theoretically and experimentally the differences between bloat control mechanisms for genetic programming. In our investigations, we consider simple tree-based genetic programming already analyzed by Durrett et al. (2011); Neumann (2012); Wagner and Neumann (2012); Nguyen et al. (2013) and Neumann et al. (2011) for problems with isolated program semantics. A possible solution is represented by a syntax tree. The tree's inner nodes are labeled by function symbols from a set F, and the tree's leaves are labeled by terminals from a set T.

Even though many GP algorithms allow complex functions for the inner nodes, we restrict the set of functions to the single binary function "join" J. Effectively, we use Js to achieve variable-length lists by concatenating leaf nodes.

The problem that we use as the basis for our investigations is a classical problem from computational complexity analysis, namely, the sorting problem SORTING. Scharnow et al. (2004) considered SORTING as an optimization problem, where different functions measure the sortedness of a permutation of given input elements. They discovered that different fitness functions lead to problems of different difficulties.

It is important to note that in contrast to WORDER and WMAJORITY (analyzed in previous articles), the SORTING problem cannot be split into subproblems that can be solved independently. These dependencies have a significant impact on the time needed to solve the problem.

We analyze our algorithms on different measures of sortedness. The problem SORTING can be stated as follows. Given a totally ordered set (of terminals) $T = \{1, \dots, n\}$ of n elements, the task is to find a permutation π_{opt} of the elements of T such that

$$\pi_{\text{opt}}(1) < \pi_{\text{opt}}(2) < \cdots < \pi_{\text{opt}}(n)$$

holds, where $<$ is the order on T. Without loss of generality, we assume $\pi_{\text{opt}} = id$, meaning that $\pi_{\text{opt}}(i) = i$ for all i, throughout our analyses.

The set of all permutations π forms a search space that has already been investigated by Scharnow et al. (2004) for the analysis of permutation-based evolutionary algorithms. The authors of that article investigated SORTING as an optimization problem where the goal was to maximize the sortedness (or equivalently, minimize the unsortedness) of a given permutation. Here, we consider the same fitness functions as introduced by Scharnow et al. (2004):

INV (π) measures the number of pairs of neighboring elements in correct order (larger values are better);

HAM (π) measures the number of elements that are at their correct position, which is the number of indices i such that $\pi(i) = i$ (larger values are better);

Algorithm 1 Derivation of $F(X)$ for SORTING

1 Generate π by parsing the tree X in order and by adding an element to π only if it is not yet in π;
2 Return $F(\pi)$;

RUN (π) measures the number of maximal sorted blocks, which is the number of indices i such that $\pi(i+1) < \pi(i)$ plus 1 (smaller values are better);

LAS (π) measures the length of the longest ascending subsequence within π of elements (larger values are better);

EXC (π) measures the smallest number of pairwise exchanges in π in order to sort the sequence (smaller values are better);

Given a tree X, we determine the permutation π that it represents according to Algorithm 1. Once we have seen an element during an in-order parse, we skip its duplicates. This is necessary, as the resulting sequence of elements for which we determine its sortedness should contain each element at most once. Note also that a single target permutation can be represented by many different trees.

Note that EXC(π) can be computed in linear time because of the cycle structure of permutations. The sequence is sorted if and only if it has n cycles. Otherwise, it is always possible to increase the number of cycles by exchanging an element that is not sitting at its correct position with the element that is currently sitting there. For any given permutation π consisting of $n-k$ cycles, EXC(π) $= k$.[2]

We investigate the five listed measures for variable-length evolutionary algorithms. Consequently, we might have to deal with incomplete permutations, as not all elements have to be contained in a given individual. Most measures can also be used for incomplete permutation, but we have to make sure that complete permutations always obtain a better fitness than incomplete ones, so that the sortedness measure guides the algorithm from incomplete permutations to complete ones. Therefore, we use the sortedness measures as described and use the following special fitness assignments that enforce these properties:

INV (π) is the number of pairs in order, except INV (π) $= 0$ if $|\pi| = 0$, and INV (π) $=$ 0.5 if $|\pi| = 1$;

RUN (π) $= n + 1$ if $|\pi| = 0$, otherwise RUN (π) $= b + m$ is the sum of the number of maximal sorted blocks b, and the number of elements missing $m = n - |\pi|$;

If $|\pi| \leq n$ then EXC (π) $= e + m + 1$, else EXC (π) $= e$, where e is the number of necessary exchanges, and $m = n - |\pi|$ the number of missing elements.

Note that e can be computed for incomplete permutations as well, as only the order $<$ on the expressed variables has to be respected. This means that the permutations $\pi_1 = (1, 4)$ and $\pi_2 = (1, 2, 3, 4)$ require no changes, but EXC (π_1) $\neq$ EXC (π_2), as the number of missing elements differs.

[2]A cycle can be determined by starting at a position in the permutation and then following the positions by using the values at the positions as indices for the next position. We recommend that the reader write down a random permutation underneath the sorted sequence.

Algorithm 2 HVL-Prime mutation operator

1 Mutate tree Y by applying HVL-Prime k times; each time choose either insert, substitute, or delete uniformly at random.
2 **if** *Insert* **then**
3 Choose a variable $u \in L$ uniformly at random and select a node $v \in Y$ uniformly at random. Replace v by a join node whose children are u and v, in which their orders are chosen randomly.
4 **if** *Substitute* **then**
5 Replace a randomly chosen leaf $v \in Y$ by a randomly chosen terminal $u \in T$.
6 **if** *Delete* **then**
7 Choose a leaf node $v \in Y$ randomly with parent p and sibling $u \neq v$. Replace p by u and delete p and v.

For example, for a tree X with $\pi = (2, 3, 4, 5, 1, 6)$ and $n = 7$, the sortedness results are HAM $(X) = 1$, RUN $(X) = 2 + 1 = 2$, and EXC $(X) = 4 + 1 + 1 = 6$.

MO-INV, MO-HAM, MO-RUN, MO-LAS, and MO-EXC are variants of the described problems. They take as the second objective the complexity C of a syntax tree (computed by the number of leaves of the tree), for instance, MO-INV (X) = (INV (X), C(X)). Optimization algorithms can then make use of this in order to deal with the bloat problem: given two solutions with identical fitness value, the algorithms can prefer the solution of lower complexity.

3 Algorithms

In this article, all GP algorithms use only the HVL-Prime as the mutation operator to generate new solutions. HVL-Prime is a variant of O'Reilly's HVL mutation operator (O'Reilly, 1995; O'Reilly and Oppacher, 1994) and it is motivated by minimality rather than by problem-specific operations. HVL-Prime produces a new tree by making changes to the original tree via three basic operators: insertion, deletion, and substitution (see Algorithm 2). In each iteration of the algorithms, k mutations are applied to the selected solution. For the single-operation variants of the algorithms, $k = 1$ holds. For the multioperation variants, the number of operations performed is drawn each time from the distribution $k = 1 + Pois(1)$, where $Pois(1)$ denotes the Poisson distribution with mean 1.

The algorithm (1+1)-GP* that we investigate first has no explicit mechanism to control bloat whatsoever. The only feature that can potentially prevent the solution's size from becoming too large is that only strict fitness improvements are accepted. Thus, the maximum solution size is limited based on the size of the initial tree and by the number of possible improvements that can be performed.[3]

The single-objective variant called (1+1)-GP*-single (see Algorithm 3) starts with an initial solution X and produces in each iteration a single offspring Y by applying the mutation operator HVL-Prime, given in Algorithm 2 with $k = 1$. This means that it is a stochastic hill-climber that explores its local neighborhood. In the case of maximization,

[3]The naming of our GP variants follows the conventions often used in the computational complexity analysis of evolutionary algorithms: An asterisk indicates that a strict fitness improvement over the old solution is required in order for the new solution to replace the current solution.

Algorithm 3 (1+1)-GP*-single for maximization

```
Choose an initial solution X;
repeat
    Set Y := X;
    Apply the mutation operator (given in Algorithm 2) with k = 1 to Y;
    if f(Y) > f(X) then set X := Y;
```

Algorithm 4 (1+1)-GP-single for maximization

```
Choose an initial solution X;
repeat
    Set Y := X;
    Apply the mutation operator (given in Algorithm 2) with k = 1 to Y;
    if f(Y) ≥ f(X) then set X := Y;
```

Algorithm 5 SMO-GP

```
Choose an initial solution X;
Set P := {X};
repeat
    Choose X ∈ P uniformly at random;
    Set Y := X;
    Apply mutation to Y;
    if {Z ∈ P | Z ⪰ Y} = ∅ then set P := (P \ {Z ∈ P | Y ≻ Z}) ∪ {Y};
```

Y replaces X if $f(Y) > f(X)$ holds. Minimization problems are tackled in the analogous way.

The single-objective variant called (1+1)-GP (see Algorithm 4 for the single-mutation variant) is identical to the just described (1+1)-GP* with the exception that in the case of maximization Y replaces X if $f(Y) \geq f(X)$ holds. Again, minimization problems are tackled in the analogous way. As a consequence of the relaxed acceptance condition, the complexity of the solution can increase as long as the fitness does not decrease. Thus, (1+1)-GP has no mechanism to prevent bloat whatsoever.

In order to introduce the parsimony pressure to (1+1)-GP, where in case of identical fitnesses the solution of lower complexity is preferred, we employ the multiobjective variants of the presented sortedness measures, for example, MO-INV. Without loss of generality, we assume that the complexity C is to be minimized and all fitness functions F except RUN and EXC are maximized.[4] In the parsimony approach, we optimize the defined multiobjective fitness functions MO-F$(X) = (F(X), C(X))$ with respect to the lexicographic order, that is, MO-F$(X) \geq$ MO-F(Y) is true iff

$$F(X) > F(Y) \vee (F(X) = F(Y) \wedge C(X) \leq C(Y)) .$$

As the last algorithm, we consider the simple evolutionary multi-objective genetic programming algorithm (SMO-GP, see Algorithm 5) introduced by Neumann (2012)

[4]The notions can be easily adjusted to other minimization/maximization problems.

and motivated by the SEMO algorithm for fixed-length representations by Laumanns et al. (2004). Variants of SEMO have been frequently used in the runtime analysis of evolutionary multiobjective optimization for fixed-length representations (see Giel, 2003; Neumann and Wegener, 2005; Friedrich et al., 2010; Giel and Lehre, 2010; Neumann and Witt, 2010).

In this multiobjective variable-length algorithm, both criteria F and C are equally important. In order to compare two solutions, we consider the classical Pareto dominance relations:

- A solution X *weakly dominates* a solution Y (denoted by $X \succeq Y$) iff $(F(X) \geq F(Y) \wedge C(X) \leq C(Y))$.
- A solution X *dominates* a solution Y (denoted by $X \succ Y$) iff $((X \succeq Y) \wedge (F(X) > F(Y) \vee C(X) < C(Y))$.
- Two solutions X and Y are called *incomparable* iff neither $X \succeq Y$ nor $Y \succeq X$ holds.

A solution that is not dominated by any other solution in the search space is called a Pareto optimal solution. The set of all such Pareto optimal solutions forms the Pareto optimal set, and the set of all corresponding objective vectors forms the Pareto front. In multiobjective optimization, the classical goal is to compute a Pareto optimal solution for each objective vector of the Pareto front. Or, if the Pareto front is too large, the goal then is to find a representative subset of the front—the definition of "representative" depends on the investigator's preference.

SMO-GP is a population-based approach that starts with a single solution. During the optimization run it maintains a set of nondominated solutions. This set constantly approximates the true Pareto front, that is, the set of optimal trade-offs between fitness and complexity. In each iteration, it picks one solution uniformly at random and produces one offspring Y by mutation. Y is introduced into the population iff it is not weakly dominated by any other solution in P. If Y is added to the population, all individuals that are dominated by Y are discarded.

Similar to the previously introduced algorithms, SMO-GP-single uses the mutation operator HVL-Prime with $k = 1$. We also consider SMO-GP-multi, which differs from SMO-GP-single by choosing k according to $1 + Pois(1)$.

We analyze the introduced single-objective algorithms in terms of the number of iterations of their repeat loops until they have produced an optimal solution, that is, a solution of maximal or minimal fitness value for the first time. The expected number of iterations to achieve this goal is called the expected optimization time of the algorithm. Considering multiobjective algorithms, the expected optimization time refers to the expected number of iterations of the repeat loop until the population includes for each Pareto optimal objective vector a corresponding solution.

4 Standard Approach without Bloat Control

We begin our analyses with the theoretical investigation of (1+1)-GP (see Algorithm 3), which has no mechanism to control bloat. The only feature that can potentially prevent the solution size from becoming too large is that only strict fitness improvements are accepted. Thus, the maximum solution size is limited based on the size of the initial

solution, the increase in complexity per improvement, and the total number of fitness improvements during the run of the algorithm.

Recall that the single-objective variant called (1+1)-GP*-single starts with an initial solution X and produces in each iteration a single offspring Y by applying the mutation operator given in Algorithm 2 with $k = 1$. This means that it is a stochastic hill-climber that explores its local neighborhood. In the case of maximization, Y replaces X if $f(Y) > f(X)$ holds. Minimization problems are tackled in the analogous way.

4.1 Upper Bound

In this section we analyze the performance of our (1+1)-GP* variants on one of the fitness functions introduced in Section 3.

We exploit a similarity between our variants and evolutionary algorithms to obtain an upper bound on the time needed to find an optimal solution. We use the method of *fitness-based partitions*, which was originally introduced for the analysis of elitist evolutionary algorithms (see, e. g., Wegener, 2002), where the fitness of the current search point can never decrease. Although the HVL-Prime operator is complex, we can obtain a lower bound on the probability of making an improvement considering fitness improvements that arise from the HVL-Prime suboperations insertion and substitution. In combination with fitness levels defined individually for the sortedness measures, this gives us the runtime bounds in this section.

We denote by $T_{\max}$ the maximum size of the tree during the run of the algorithm and show the following theorem.

THEOREM 1: *The expected optimization time is $O(n^3 T_{\max})$ for (1+1)-GP*-single and (1+1)-GP*-multi, using INV as the sortedness measure, where n is the number of elements that are to be sorted.*

PROOF: The proof is an application of the fitness-based partitions method. Based on the observation that $n \cdot (n-1)/2 + 1$ different fitness values are possible, we define the fitness levels $A_0, \ldots, A_{n\cdot(n-1)/2}$ with

$$A_i = \{\pi \mid \mathrm{INV}(\pi) = i\},$$

meaning that trees are assigned to the same fitness level if the number of pairs of elements in the in-order parsed sequence of leaves (according to Algorithm 1) is identical.

As there are at most $n \cdot (n-1)/2$ advancing steps between fitness levels to be made, the total expected runtime is upper bounded by the sum over all times needed to make such steps.

We bound the times by investigating the case when only a particular insertion of a specific leaf at its correct position achieves an increase of the fitness.[5] For this particular insertion, we consider the lexicographically smallest pair (i, j), $i < j$, which is currently incorrect: putting i directly before j makes this pair correct. We now have to show that this does not make incorrect any other pair that was previously correct. Assume there is a pair (k, l), $k < l$, that was previously correct and that has become incorrect because of the insertion of i. As only i is moved, $l = i$ has to hold, but we can show that this cannot be the case. k has to be smaller than j; otherwise the pair cannot become incorrect. Thus, $k < i < j$ has to hold because $k < l$ and $i < j$ and because of our assumption $l = i$. (k, j) was correct before the insertion, so it has to be lexicographically smaller than

[5]For example, the tree with the sequence of leaves (when parsed in order) $l = (n, n, 1, 2, \ldots, n-1)$ can only be improved (in a single HVL-Prime step) by inserting a leaf labeled 1 at the leftmost position.

(i, j). Therefore k is before j in the list of expressed leaf nodes. As i is placed directly before j and therefore after k, (k, l) cannot become incorrect.

The probability for HVL-Prime to perform an insertion is $\frac{1}{3}$, and the probability for the insertion to insert the new leaf at the correct position of the inner J-node is at least $\frac{1}{2}$. This, together with the probability of selecting the right element to add, which is bounded by $\frac{1}{n}$, and the probability of adding it to the right position in the tree, which is bounded by $\frac{1}{T_{\max}}$, gives us a lower bound on the probability for doing such an improvement in (1+1)-GP*-single :[6]

$$\frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{n} \cdot \frac{1}{T_{\max}} = \Omega\left(\frac{1}{nT_{\max}}\right).$$

For the multioperation variant, the probability for a single mutation operation occurring (including the mandatory one) is $1/e$, which is a constant. Thus we have an improvement with probability $\Omega\left(\frac{1}{nT_{\max}}\right)$ in the multioperation case as well. Therefore, the expected optimization time for both algorithms is upper bounded by

$$\sum_{k=0}^{n\cdot(n-1)/2} O\left(nT_{\max}\right) = O(n^3 T_{\max}).$$

□

4.2 Local Optima

In the following, we present several worst-case examples for HAM, RUN, LAS, and EXC that demonstrate that (1+1)-GP* can get stuck in local optima during the optimization process. This shows that evolving a solution with this GP system is much harder than working with the permutation-based EA presented by Scharnow et al. (2004), where only the sortedness measure RUN leads to an exponential optimization time.

We study worst-case solutions that are hard to improve by our algorithms. In the following, we write down such solutions by the order of the leaves in which they are visited by the in-order parse of the tree. We restrict ourselves to the case where we initialize with a tree of size linear in n and show that even this leads to difficulties for the previously mentioned sortedness measures. Note, that a tree of size linear in n is necessary to represent a complete permutation of the given input elements.

For RUN and LAS, we investigate the initial solution I_{w1}, defined as

$$I_{w1} = (\underbrace{n, n, \ldots, n}_{n+1 \text{ instances of } n}, 1, 2, 3, \ldots, n),$$

and show that it can be hard to achieve an improvement.

THEOREM 2: *Let I_{w1} be the initial solution. Using the sortedness measures RUN and LAS, the expected optimization time of (1+1)-GP*-single is infinite and that of (1+1)-GP*-multi is $e^{\Omega(n)}$.*

PROOF: We consider (1+1)-GP*-single first. It is clear that with a single HVL-Prime application, it is possible to remove only one of the leftmost ns. To improve the sortedness based on RUN or LAS, all leftmost $n + 1$ leaves have to be removed at once. Clearly, (1+1)-GP*-single cannot do this, which results in an infinite runtime.

[6]For example, for the new element to be inserted as the leftmost node of the tree, insertion has to be chosen, then the old leftmost node has to be chosen, and then the new node has to be placed as the left sibling of the old leftmost node, not as its right sibling.

Similarly, (1+1)-GP*-multi can only improve the sortedness if it removes the leftmost $n+1$ leaves. Hence, in order to successfully improve the sortedness, at least $n+1$ suboperations have to be performed, assuming that we delete one of the leftmost ns each time. As the number of suboperations per mutation is distributed as $1 + Pois(1)$, the Poisson random variable has to take a value of at least n. Thus, the probability for such a fitness-improving step is $e^{-\Omega(n)}$, with the expected waiting time for such a step being $e^{\Omega(n)}$. □

Similarly, we consider the tree I_{w2}, defined as

$$I_{w2} = (\underbrace{n, n, \ldots, n}_{n+1 \text{ instances of } n}, 2, 3, \ldots, n-1, 1, n),$$

and show that it is hard to improve the sortedness when using the measures HAM and EXC.

THEOREM 3: *Let I_{w2} be the initial solution. Using the sortedness measures HAM and EXC, the expected optimization time of (1+1)-GP*-single is infinite and of (1+1)-GP*-multi is $e^{\Omega(n)}$.*

PROOF: We use similar ideas as in the previous proof. Again, it is not possible for (1+1)-GP*-single to increase the sortedness in a single step, as all $n+1$ leftmost leaf nodes need to be deleted in order for the rightmost n to become expressed. In addition, a leaf labeled 1 has to be inserted at the beginning, or alternatively, one of the $n+1$ leaves labeled n has to be replaced by a leaf labeled 1. This results in a minimum number of $n+1$ suboperations that have to be performed by one HVL-Prime application, leading to the lower bound of $e^{\Omega(n)}$ for (1+1)-GP*-multi. □

5 Parsimony Approach

In this section we consider simple variable-length evolutionary algorithms using the parsimony approach. The single-objective variant called (1+1)-GP is identical to the previously investigated (1+1)-GP* with the exception that in the case of maximization, Y replaces X if $f(Y) \geq f(X)$.

Wagner and Neumann (2012) showed that the optimization time of (1+1)-GP-single on MO-EXC, MO-RUN, and MO-HAM is infinite when initialized with specific solutions.

In the following, we add to these results by proving polynomial runtime bounds for the other functions. The idea behind the proof of the expected polynomial optimization time on MO-LAS is as follows. Given a tree T with its tree size of T_{init} and its sortedness $\text{LAS}(T) = k < n$. For such a tree, we always have at least one of the following two ways to create a new tree that is accepted. First, we can improve the sortedness by extending the longest ascending sequence. Or, second, we can reduce the size of the tree, if the tree has more than k leaves. If the latter is the case, we can trim the number of leaves down to k, thus eliminating blocking elements and duplicates, and then we can build up the sought permutation. Thus, we can now deal with trees such as I_{w1} (see Section 4.2) that have previously been problematic.

THEOREM 4: *The expected optimization time of (1+1)-GP-single on MO-LAS is $O\left(T_{\text{init}} + n^2 \log n\right)$.*

PROOF: We consider two phases. First, we show that we arrive at a tree with fitness k and k leaves after $O\left(T_{\text{init}} + n \log n\right)$ steps. Then we analyze the time needed to get from there to the optimal solution.

1. Phase. Initially, let $\mathrm{LAS}(T) = k$ be the fitness of the current tree T with s leaves. Then, the fitness distance to the desired tree size is $d = s - k$. As the probability for HVL-Prime to perform a deletion is $\frac{1}{3}$, the probability to reduce the size via a deletion in a single mutation step is at least

$$\frac{1}{3} \cdot \frac{s-k}{s} = \frac{1}{3} \cdot \frac{d}{d+k} \geq \frac{1}{3} \cdot \frac{d}{d+n},$$

where the term $\frac{s-k}{s}$ comes from the fact that we need to select one of the redundant elements. Note that d cannot increase: for d to increase, k would have to decrease, which is impossible, as the primary objective is the maximization of the LAS value. Alternatively, d could increase if s increases. However, the tree size can only increase if the last accepted step increased the sortedness as well. In a single step, if s increases by 1, then k had to increase by 1 as well, which leaves the distance $s - k = d$ unchanged.

Now, with the fitness-based partitions method over the distance d, and the fitness levels $A_0, \ldots, A_{T_{\text{init}}}$ with

$$A_i = \{T \mid i = T_{\text{init}} - d = T_{\text{init}} - |T| + \mathrm{LAS}(T)\},$$

we can bound the expected runtime for this first phase:

$$\begin{aligned}
\sum_{d=1}^{T_{\text{init}}} 3\frac{d+n}{d} &= 3\sum_{d=1}^{n} \frac{d+n}{d} + 3\sum_{d=n+1}^{T_{\text{init}}} \frac{d+n}{d} \\
&\leq 3\sum_{d=1}^{n} \frac{d+n}{d} + 3\sum_{d=n+1}^{T_{\text{init}}} 2 \\
&= O\left(n \log n + T_{\text{init}}\right).
\end{aligned}$$

2. Phase. Next, we investigate the time needed in the second phase to arrive at the optimum. Therefore, we again apply the described fitness-based partitions method. We define the fitness levels $A_1, \ldots, A_n$ with $A_i = \{T \mid \mathrm{LAS}(T) = i\}$. As there are at most $n-1$ advancing steps between fitness levels to be made, the total expected runtime is upper bounded by the sum over all expected times needed to make such steps.

After the initial trimming phase, we do not have any blockages that prevent elements from being expressed at their correct positions. Therefore, the existing longest ascending sequence can be extended by inserting *any* of the $n-k$ unblocked elements that are missing in the sequence into its correct position. The probability for a single of such an insertion to happen is at least $\frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{n} \cdot \frac{n-k}{n} = \frac{1}{6} \cdot \frac{n-k}{n^2}$. Consequently, the expected runtime of the second phase can then be bounded from above by

$$\sum_{k=1}^{n-1} 6\frac{n^2}{n-k} = 6n\sum_{k=1}^{n-1} \frac{n}{n-k} = O\left(n^2 \log n\right).$$

Hence, the expected optimization time of the algorithm is $O\left(T_{\text{init}} + n^2 \log n\right)$. □

THEOREM 5: *The expected optimization time of (1+1)-GP-single on MO-INV is upper bounded by* $O\left(T_{\text{init}} + n^5\right)$.

PROOF: We draw upon results from Theorems 1 and 4. First, after $O\left(n \log n + T_{\text{init}}\right)$ steps, we arrive at a nonredundant tree. Next, as we can have at most n^2 fitness-improving insertions, the maximum tree size $T_{\max}$ is bounded by $O\left(n + n^2\right)$ after the

initial trimming phase. Consequently, the probability for a fitness-improving mutation is bounded by $\Omega\left(\frac{1}{n^3}\right)$. Thus, we can now bound the overall optimization time by

$$\begin{aligned} & O\left(n \log n + T_{\text{init}}\right) + \sum_{k=0}^{n\cdot(n-1)/2} O\left(n^3\right) \\ &= O\left(n \log n + T_{\text{init}}\right) + O(n^5) \\ &= O\left(T_{\text{init}}\right) + O(n^5). \end{aligned}$$

□

Achieving a similar bound for the multimutation variant is not as easy, since the insertion of a missing element (i.e., a fitness improvement) may be accompanied by the insertion of many elements that are already present. Because of the Poisson-distributed number of operations performed by HVL-Prime within (1+1)-GP-multi, the algorithm's *typical* local behavior is difficult to predict.

Therefore, we take an alternative approach, looking at a polynomial-sized sequence of steps $t = poly(n)$. Let T_{init} be a tree with size $|T_{\text{init}}| = poly(n)$. The failure probability for inserting at most n^ϵ in a single HVL-Prime operation is $e^{-\Omega(n^\epsilon)}$. Furthermore, given any initial tree, we can have at most n improvements of the sortedness when the measurements LAS and EXC are used. Now, we compute a bound of the tree size. Looking at n mutations that increase the fitness, the failure probability for adding at most $nn^\epsilon = n^{1+\epsilon}$ leaves in t time steps is exponentially small: $te^{-\Omega(n^\epsilon)} = e^{-\Omega(n^\epsilon)}$. Thus, the tree size does not exceed $T_{\max} = T_{\text{init}} + n^{1+\epsilon}$ within $t = poly(n)$ time steps, with high probability.

THEOREM 6: *Let $\epsilon > 0$ be a constant. The optimization time of (1+1)-GP-multi on MO-LAS is $O\left(T_{\text{init}} + n^2 \log n\right)$, with probability $1 - o(1)$.*

PROOF: We split the proof into two parts: first, we bound the total time needed for deletions during a run, and second, we investigate the time needed to perform the necessary insertions to find the optimal solution.

First, given a solution where k_m elements have to be removed in order to arrive at a nonredundant tree after the mth fitness-increasing insertion. In the following, let i be the number of redundant elements in the tree, and let j be the number of nonredundant elements in the tree.

Stage 1, $i \geq n + 1$. As the probability for a single operation is $\frac{1}{e}$, the probability for the deletion of a single redundant element at any time is lower bounded by

$$\frac{1}{3e}\frac{i}{i+j} \geq \frac{1}{3e}\frac{i}{i+n} \geq \frac{1}{3e}\frac{1}{2} = \frac{1}{6e}.$$

Then, the expected time to delete k_m elements is upper bounded by $6ek_m$. Furthermore, as we know that we can delete at most $T_{\max}$ leaves over a full optimization run, $\sum_{i=1}^{n} k_i \leq T_{\max}$. Thus, we can bound the expected time needed for *all* deletions (when $i \geq n + 1$) by $6eT_{\max}$.

Let $X_1, \ldots, X_d$ be independent random variables taking value 1 with $Prob(X_i = 1) = \frac{1}{6e}$ if an element is deleted (in time step $1 \leq t \leq d$), and 0 otherwise. With Chernoff's

inequality[7] (with $\delta = 1$) we get that

$$Prob\,(X \geq 12eT_{\max}) = Prob\left(X \geq 12e(T_{\text{init}} + n^{1+\epsilon})\right) \leq e^{-2e(T_{\text{init}}+n^{1+\epsilon})} \leq e^{-\Omega\left(n^{1+\epsilon}\right)}.$$

Stage 2, $i \leq n$. To bound the number of steps, we apply the technique of multiplicative drift with tail bounds (see Definition 1 and Theorem 1 in Doerr and Goldberg, 2010).

In our situation, $\Phi(x) = i$ is a feasible ν-drift function on the number of redundant elements (with implicit constant $\delta = 1$). For the optimal solutions ("no redundant elements left") $\Phi(x) = 0$ holds as required, $\Phi(x) \geq 1$ holds for all nonoptimal solutions, and $E[\Phi(x_{\text{new}})] \leq \left(i - \frac{i}{6en}\right) = \left(1 - \frac{1}{\nu(n)}\right)\Phi(x)$. Thus, $\nu(n) = 6en$ and $\delta = 1$. Consequently, we get that the time needed for all deletions (when $i \leq n$) during a run exceeds $6en(\ln n + n \ln n)$ with probability at most n^{-c}. As these deletion phases take place at most n times, the resulting overall deletion time does not exceed $O(n^2 \log n)$ with probability $1 - n^{-c+1} = 1 - o(1)$.

Next, we consider the time necessary to perform the insertions of the missing elements once the insertion was unblocked. We again apply the multiplicative drift with tail bounds. Note that the situation is very similar: instead of reducing the number of redundant elements, we are now reducing the number of missing elements.

Let j be the number of elements currently missing. As the probability for a single operation is $\frac{1}{e}$, the probability for a single insertion of a missing element to happen at the required position is lower bounded by $\frac{1}{3e}\frac{1}{2n}\frac{j}{n} = \frac{j}{6en^2}$. With $E[\Phi(x_{\text{new}})] \leq \left(j - \frac{j}{6en^2}\right) = \left(1 - \frac{1}{\nu(n)}\right)\Phi(x)$, we get $\nu(n) = 6en^2$ and $\delta = 1$. Consequently, by applying Theorem 1 from Doerr and Goldberg (2010), we get that the time needed for all insertions during a run exceeds $6en^2(\ln n + n \ln n)$ with probability at most n^{-c}. Thus, the resulting overall time needed for all insertions does not exceed $O(n^2 \log n)$ with probability $1 - n^{-c} = 1 - o(1)$. □

6 Multiobjective Approach

In the following, we consider the simple evolutionary multi-objective genetic programming algorithm (SMO-GP, see Algorithm 5) introduced by Neumann (2012), which modifies the original SEMO algorithm of Laumanns et al. (2004) to work with fixed-length representations.

Let us recall that SMO-GP is a population-based approach that is initialized with a single solution. During the run, it keeps in each iteration the set of nondominated solutions obtained so far. This set of solutions constantly approximates the true Pareto front, namely, the set of optimal trade-offs between fitness and complexity. In each iteration, it picks one solution uniformly at random and produces one offspring Y by mutation. Y is introduced into the population iff it is not weakly dominated by any other solution in P. If Y is added to the population, all individuals that are dominated by Y are discarded.

[7] Let random variables $X_1, \ldots, X_n$ be independent random variables taking on values 0 or 1. Further, assume that $P(X_i = 1) = p_i$. Then, if we let $X = \sum_{i=1}^{n} X_i$ and $E[X]$ be the expectation of X, then the following bound holds: $P(X \geq (1+\delta)E[X]) \leq e^{-E[X]\delta^2/3}$, $0 < \delta \leq 1$.

In the following, we analyze the expected number of iterations before the set of nondominated solutions becomes the true Pareto front. We call this the *expected optimization time* of SMO-GP algorithms.

For arbitrary optimization problems, the following lemma bounds the expected time needed for the populations to include the empty solution (i.e., the empty tree):

LEMMA 1 (Neumann, 2012): *Let I_{init} be the size of the initial solution and k be the number of different fitness values of a problem F. Then the expected time until the population of SMO-GP-single and SMO-GP-multi applied to MO-F contains the empty solution is $O(kI_{\text{init}})$.*

THEOREM 7: *The expected optimization time of SMO-GP-single and SMO-GP-multi is $O(n^2 I_{\text{init}} + n^5)$ on MO-INV, and $O(nI_{\text{init}} + n^3 \log n)$ on MO-LAS.*

PROOF: First, as INV has $n(n-1)$ different fitness values, using Lemma 1, the empty solution is produced after an expected number of $O\left(n^2 I_{\text{init}}\right)$ steps. First, note that each Pareto optimal solution with complexity $2i-1$ has an INV value of $\sum_1^{i-1} i$, if $i \geq 2$.[8]

Second, we bound the time needed to discover the whole Pareto front once the empty solution is introduced into the population. Assume that the population contains all Pareto optimal solutions with complexities $2j-1$, $1 \leq j \leq i$. Then, a population that includes all Pareto optimal solutions with complexities $2j-1$, $1 \leq j \leq i+1$ can be achieved by producing a solution Y that is Pareto optimal and that has complexity $2(i+1)-1$. Y can be obtained from a Pareto optimal solution X with $C(X) = 2i-1$ by inserting an element that increases the INV value by $i-1$. This operation produces from a solution of complexity $2i-1$ a solution of complexity $2(i+1)-1 = 2i+1$, as one leaf node and one inner node are added.

Based on this idea we can bound the expected optimization time once we can bound the probability for such steps to happen. With probability at least $\frac{1}{n(n-1)/2+1}$ it is possible to choose X, as the population size is upper bounded by $n(n-1)/2+1$. Next, a single mutation operation happens with probability at least $1/e$, and the inserting operation of HVL is chosen with probability $1/3$. The probability to select one of the missing elements is at least $1/n$. However, the correct position for such a randomly chosen element has to be chosen in order to produce a solution that is Pareto optimal and of complexity $i+1$. This probability is at least $1/2 \cdot 1/n$, as the number of leaf nodes is bound by n, and the probability to insert as the correct child of the newly introduced inner node is at least $1/2$. Thus, the total probability of such a generation can be bounded by $\frac{1}{n(n-1)/2+1} \cdot \frac{1}{3e} \cdot \frac{1}{2n} \cdot \frac{1}{n}$.

Therefore, as only n Pareto optimal improvements are possible once the empty solution has been introduced into the population, the expected time until all Pareto optimal solutions have been generated is bounded by

$$\sum_{i=0}^{n} \left(\frac{1}{n(n-1)/2+1} \cdot \frac{1}{3e} \cdot \frac{1}{2n} \cdot \frac{1}{n} \right)^{-1} = 6en^5 = O(n^5).$$

Similarly, we can prove an upper bound for MO-LAS. First, note that each Pareto optimal solution with LAS value i represents a perfectly sorted permutation of i elements. Next, after an expected number of $O\left(nI_{\text{init}}\right)$ steps the empty solution is produced, as only n different LAS values are possible. As before, we assume that the population already contains all solutions that are Pareto optimal and of complexities $2j-1$,

[8]For the sake of readability, the special cases for $i = 0$ and $i = 1$ are omitted in the following.

Table 1: Summary of computational complexity bounds for single-objective variants.

	(1+1)-GP*, F(X)		(1+1)-GP, F(X)
F(X)	single	multi	single/multi
INV	$O(n^3 T_{\max})$ *	$O(n^3 T_{\max})$ *	
LAS	∞ *	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$*	
HAM	∞ *	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$*	?
EXC	∞ *	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$*	
RUN	∞ *	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$*	

The question mark indicates combinations for which we do not know any bounds. Asterisks indicate the bounds presented in this article. $T_{\max}$ denotes the size of the largest tree at any stage during the evolution of the algorithm.

$1 \leq j \leq i$. Then, the minimally larger population that includes all Pareto optimal solutions with complexities $2j-1$, $1 \leq j \leq i+1$, can be achieved by inserting any of the missing $n-i$ elements into its correct position in the Pareto optimal individual X with $\mathrm{LAS}(X) = C(X) = 2i-1$.

Therefore, as only n Pareto optimal improvements are possible once the empty solution has been introduced into the population, the expected time until all Pareto optimal solutions have been found is

$$\sum_{i=0}^{n} \left(\frac{1}{n+1} \cdot \frac{1}{3e} \cdot \frac{1}{2n} \cdot \frac{n-i}{n} \right)^{-1} = 6en^2(n+1) \cdot \sum_{i=0}^{n} \frac{1}{n-i}$$
$$= O(n^3 \log n).$$

□

7 Complementary Experimental Analyses

Tables 1 and 2 summarize our theoretical findings, list existing bounds, and show open problems. As can be observed from the tables, all bounds consider tree sizes of some kind: either the size $T_{\max}$ of the largest tree or the size of the initial solution T_{init}. In particular, the runtime of (1+1)-GP*, F(X) depends on the maximum tree size $T_{\max}$, since the expected time to get to the optimal solution grows larger and larger as the tree grows in size. The runtimes of several MO-F(X) variants depend on the initial tree size T_{init}, as often the first step of the proof involves deconstructing the original solutions until a tree of size zero is found. Furthermore, it is quite striking that not too many bounds for the multioperation GP algorithms are known so far. It is also not known whether the bounds are tight or not. As the maximum tree size for (1+1)-GP and the population size for SMO-GP play a relevant role in the theoretical analyses, we focus our attention on these.

In this section we carry out experimental investigations about the runtime of different variable-length algorithms over the presented fitness functions. The purpose of this analysis is threefold:

- to complement the theoretical results with conjectures about the expected optimization times for the variants lacking a formal proof;

Table 2: Summary of computational complexity bounds for multiobjective variants.

	(1+1)-GP, MO-F(X)		SMO-GP, MO-F(X)
F(X)	single	multi	single/multi
INV	$O(T_{\text{init}} + n^5)$ *	?	$O\left(n^2 T_{\text{init}} + n^5\right)$ *
LAS	$O(T_{\text{init}} + n^2 \log n)$ *	$O(T_{\text{init}} + n^2 \log n)$ †*	$O(nT_{\text{init}} + n^3 \log n)$ *
HAM	∞	?	$O(nT_{\text{init}} + n^4)$
EXC	∞	?	$O(nT_{\text{init}} + n^3 \log n)$
RUN	∞	?	$O(nT_{\text{init}} + n^3 \log n)$

† indicates a bound that holds with probability $1 - o(1)$. Question marks indicate combinations for which we do not know any bounds. Asterisks indicate bounds presented in this article. T_{init} denotes the size of the initial tree.

- to assess the impact on the runtime of two collected measures, namely, the maximum tree size $T_{\max}$ and, for SMO-GP, the maximum population size $P_{\max}$ encountered during an optimization run; and
- to give useful insight for guiding further rigorous theoretical analysis.

7.1 Experimental Setup

In our experimental investigations, we considered all the GP algorithms: (1+1)-GP on F(X), (1+1)-GP on MO-F(X), (1+1)-GP* on F(X), and SMO-GP on MO-F(X). Each GP algorithm was run in its single-mutation and multimutation variants, and we investigated problems of sizes $n = 20, 40, 60, \ldots, 200$. For the initialization of the individuals, we considered the schemes init_0 (empty tree) and init_n (tree with n leaves constructed by applying n insertion mutations at random positions on an initially empty tree). In total, our experiments spanned ten problems: INV, HAM, RUN, LAS, and EXC in their single and multiobjective variants.

We ran the experiments on Intel Xeon E5430 CPUs (2.66GHz) on Debian GNU/Linux 7 with Java SE RE 1.7. We limited the computation budget to a maximum runtime of 3 hours or 10^9 evaluations each, whichever was reached first. Furthermore, we repeated each experiment 200 times, resulting in a standard error of the mean (the standard deviation of the sampling distribution) of $1/\sqrt{200} = 7\%$. As a curiosity, the whole set of experiments took about 30 CPU-years to complete.

The complete source code of the framework is available on BitBucket (Mercurial, at https://bitbucket.org/tunnuz/gpframework), on GitHub (Git, https://github.com/tunnuz/gpframework), and on Google Code (Subversion at http://code.google.com/p/gpframework).

7.2 Experimental Analysis of the (1+1)-GP Variants

We now analyze the experimental results of the (1+1)-GP variants with respect to the maximum tree size obtained during execution and the required optimization time.

7.2.1 Tree Size

As mentioned, the known theoretical bounds for the (1+1)-GP variants depend on $T_{\max}$, the size of the largest tree encountered during the run of the algorithm. It is important to observe that the maximum solution size is not a parameter that is set in advance but rather a measure that emerges from the nature of the employed fitness function and mutation operators. In addition, the optimization can involve a degree of randomness,

which makes $T_{\max}$ (and thus bloating) extremely difficult to predict. For this reason, we investigate the maximum solution size experimentally to detect when bloat occurs within the analyzed GP algorithms. As statistics, we employ the median (the second quartile) as a measure of central tendency and the interquartile range (iqr, the distance between the first and the third quartiles) as a measure of variance.

Table 3 reports results for $n = 40, 80, 160$, and the results for the other input sizes are comparable. The missing data (—) represent experiments for specific input sizes where the algorithms did not find an optimal solution within the time or evaluations bound for more than 50% of the repetitions.[9] For the sake of clarity we recall that (1+1)-GP*, F(X) accepts a new solution only if the fitness is strictly better than the previous one, while (1+1)-GP, F(X) always accepts a solution of the same value. (1+1)-GP, MO-F(X) accepts a solution of the same fitness only if the complexity is lower.

We first analyze $T_{\max}$ for the single-operation variant, where a single mutation operator is applied at each step (upper half of Table 3). Here (1+1)-GP* and (1+1)-GP, MO-F(X) share similar tree sizes of about $2n - 1$ (sometimes $2n + 1$), which is a minimum for the optimal solution on all fitness values but INV, where (1+1)-GP, MO-F(X) obtains a tree size of about $2.3n$ on both initialization schemes and (1+1)-GP* shows a tree size close to $10n$. On the other hand, (1+1)-GP, F(X) appears cursed by bloating in all fitness functions, with tree sizes above $12n$. Nonetheless, unlike (1+1)-GP*, (1+1)-GP, F(X) seems independent on the employed initialization scheme and can reach optimal solutions for INV and LAS even with init_n where (1+1)-GP* fails. As for the interquartile range, (1+1)-GP, MO-F(X) appears to be the most stable algorithm with an iqr of zero on all fitness functions but INV. Overall, the best algorithm with respect to tree size, interquartile range, and robustness with respect to initialization schemes is (1+1)-GP with parsimony (MO-F(X) variant).

As for the multioperation variant, that is, where $k = 1 + Pois(1)$ applications of each mutation operator are executed at each step, tree sizes increase in every algorithmic variant on INV. On the other fitness functions, the negative impact of multiple operations appears especially on the F(X) variants, while (1+1)-GP, MO-F(X) is less susceptible to this parameter. Overall, the interquartile range in the tree size increases along with it. Again, with respect to tree size, interquartile range, and initialization scheme independence, the best algorithm is (1+1)-GP with parsimony (MO-F(X) variant).

7.2.2 Average Case Optimization Time

Figures 1 and 2 indicate the asymptotic behavior of the investigated measures. They show

- the *distributions* of values, represented as box plots;
- the *failure rate*, that is, the fraction of repetitions that did not make it to the end because of the imposed timeout or evaluations budget, represented in red; and
- two blue lines representing for each input size the medians of the distributions divided by some polynomial, whose interpretation gives an indication of the asymptotic behavior of the measure.

In order to deduce the asymptotic behavior of a measure, one must look at the polynomial line that is closest to constant (i.e., the most horizontal one). A horizontal line

[9] For the computation of the median, at least 50% of the independent runs need to be successful.

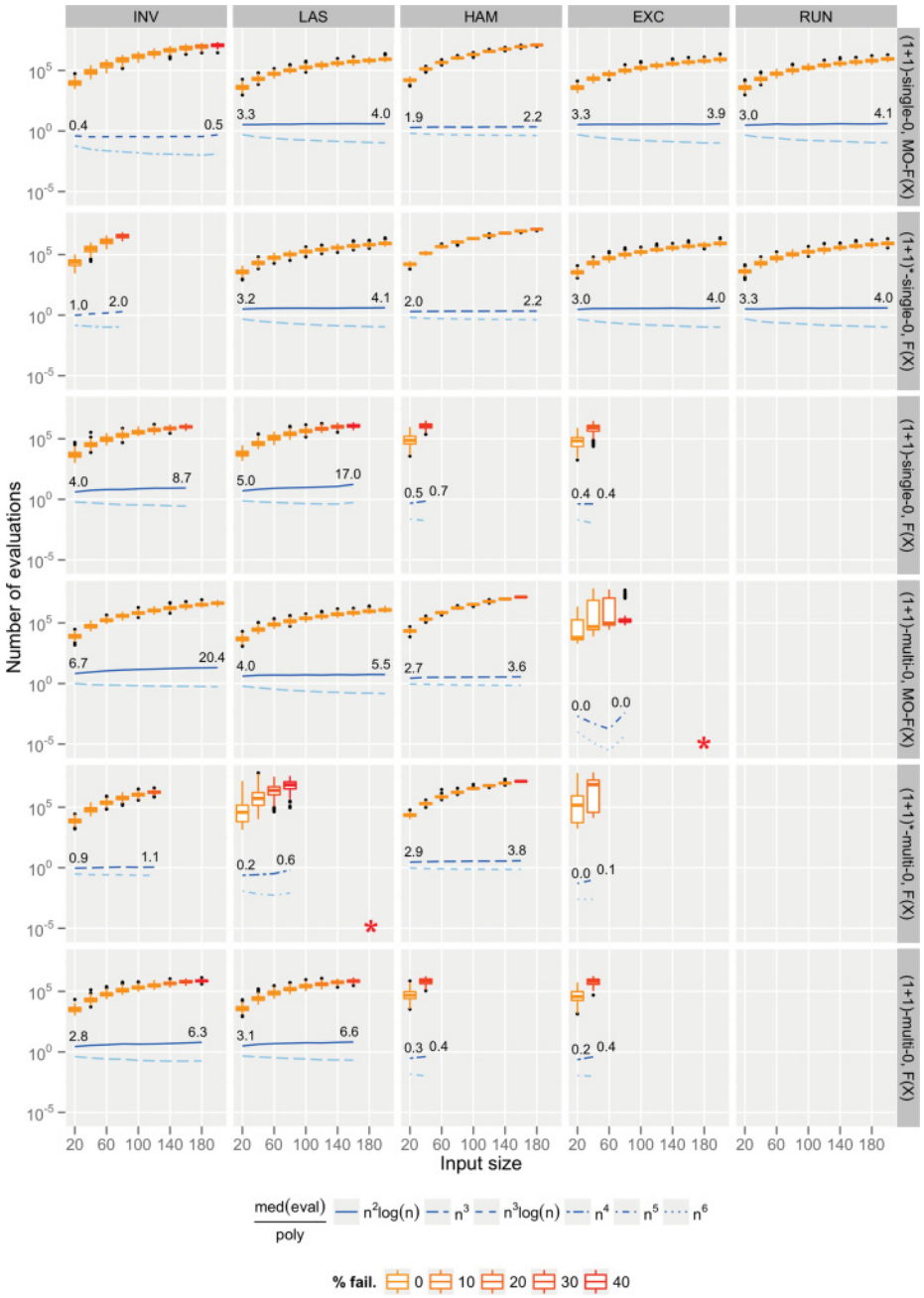

Figure 1: Box plots showing the number of evaluations required by (1+1)-GP (initialized with init_0) until the individual X_{opt} with optimal fitness was found. No data are shown when more than 50% of runs were unsuccessful. It is evident that the algorithms had problems solving even small instances of RUN. In the configuration marked with an asterisk, the method to find the upper and lower polynomials was unreliable because of inflections.

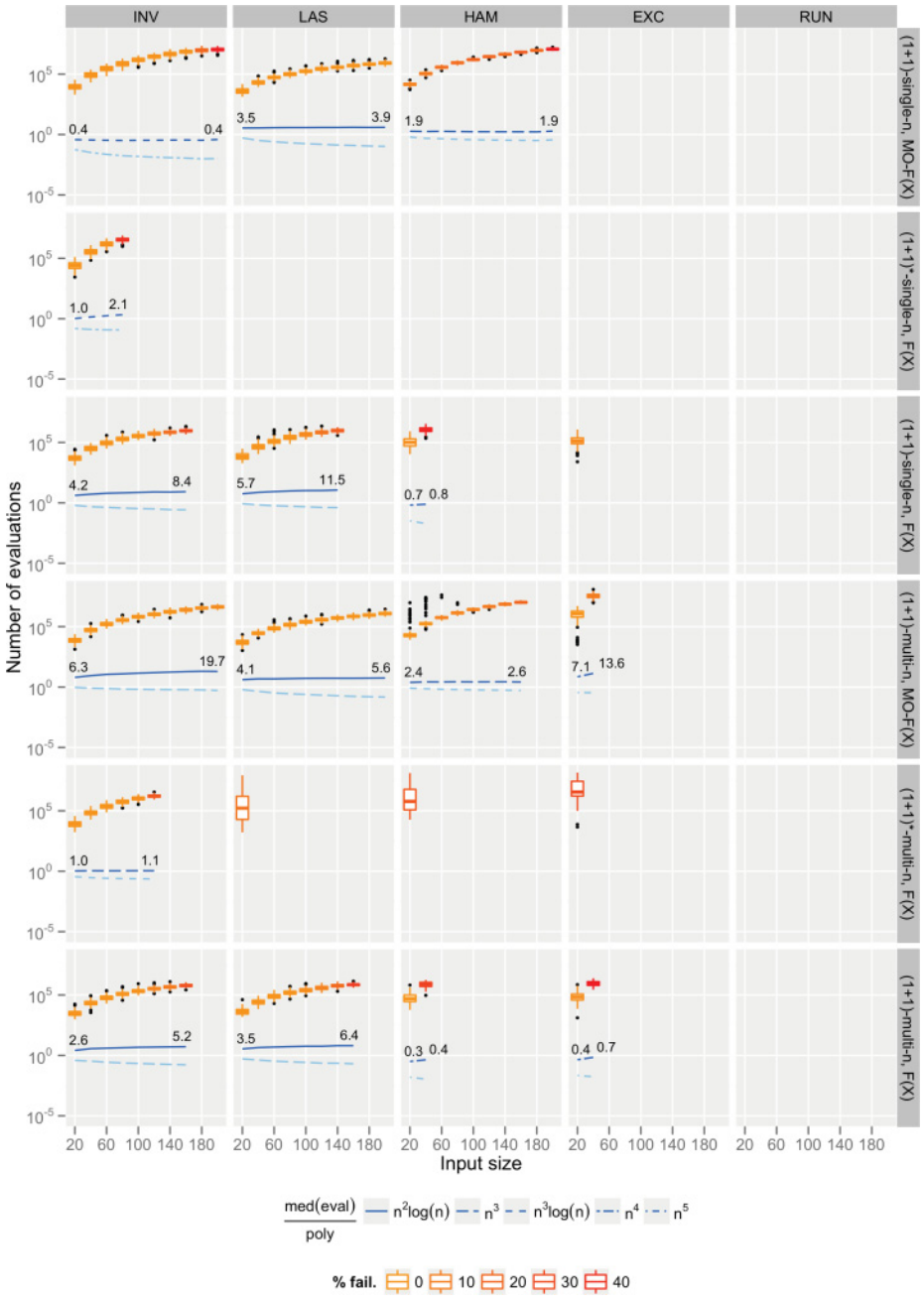

Figure 2: Box plots showing the number of evaluations required by (1+1)-GP (initialized with init_n) until the individual X_{opt} with optimal fitness was found. No data are shown when less than 50% of the runs were successful. When one compares these results with those of Figure 1, it is evident that initialization with n leaf nodes can render the problem unsolvable.

Table 3: Sizes of the largest encountered trees until the individual X_{opt} with optimal fitness is found. Shown are the median m and median interquartile ranges iqr.

k	F(X)	n	(1+1)-GP*, F(X)				(1+1)-GP, F(X)				(1+1)-GP,MO-F(X)			
			init_0		init_n		init_0		init_n		init_0		init_n	
			m	iqr	m	iqr	m	iqr	m	iqr	m	iqr	m	iqr
$k=1$	INV	40	307	46	327	33.5	528	185.5	528	202.5	95	4	97	5.5
		80	821	79	849	105	1259	472	1269	473	189	8	191	6
		160	—	—	—	—	2645	612	2688	627.5	375	10	381	14
	LAS	40	79	0	—	—	525	212	592	265.5	79	0	79	0
		80	159	0	—	—	1352	508.5	1401	526.5	159	0	159	0
		160	319	0	—	—	2670	527.5	—	—	319	0	319	0
	HAM	40	79	0	—	—	1665	1042	1672	723.5	79	0	79	0
		80	159	0	—	—	—	—	—	—	159	0	159	0
		160	319	0	—	—	—	—	—	—	319	0	319	0
	EXC	40	81	0	—	—	1573	908	—	—	81	2	—	—
		80	161	0	—	—	—	—	—	—	161	0.5	—	—
		160	321	2	—	—	—	—	—	—	321	0	—	—
	RUN	40	79	0	—	—	—	—	—	—	79	0	—	—
		80	159	0	—	—	—	—	—	—	159	0	—	—
		160	319	0	—	—	—	—	—	—	319	0	—	—
$k=1+Pois(1)$	INV	40	249	33	259	34	512	183	543	199.5	107	8	112	10
		80	611	48	627	58	1245	490	1308	435.5	213	12.5	219	14
		160	—	—	—	—	2793	733	2821	700	419	18	437	22
	LAS	40	95	10	—	—	555	276.5	560	261.5	79	2	79	2
		80	187	10.5	—	—	1334	592	1382	420.5	159	2	159	2
		160	—	—	—	—	2893	698	2789	498	319	2	319	2
	HAM	40	87	6	—	—	1767	926	1833	1042	79	2	79	2
		80	177	8	—	—	—	—	—	—	159	0	159	2
		160	353	11.5	—	—	—	—	—	—	319	2	319	2
	EXC	40	93	6	—	—	1852	1042	1964	964.5	81	0	83	2
		80	—	—	—	—	—	—	—	—	161	2	—	—
		160	—	—	—	—	—	—	—	—	—	—	—	—
	RUN	40	—	—	—	—	—	—	—	—	—	—	—	—
		80	—	—	—	—	—	—	—	—	—	—	—	—
		160	—	—	—	—	—	—	—	—	—	—	—	—

means that barring a multiplicative factor, the measure behaves like the corresponding polynomial, at least for the analyzed input sizes. The figures exclude the input sizes where a failure rate above 50% did not allow computing a reliable median (and thus obtaining a reliable estimate on the asymptotic behavior).

Figures 1 and 2 show the distributions of the required number of evaluations to reach the first optimal solution for the (1+1)-GP variants, respectively, in the init_0 and init_n initialization schemes.

By analyzing the results it can be noted that overall the init_0 initialization scheme is beneficial for (1+1)-GP*, F(X) both in single- and multioperation modes, allowing it to optimize every fitness function in single-operation mode and to reach some optima for all fitness functions except RUN for multioperation mode. To the contrary, (1+1)-GP does not seem to be influenced significantly by a particular choice of initial individuals. The performances of the two initialization schemes are identical for INV across every algorithmic variant but consistently worse for init_n in all other fitness values, at least in terms of failure rate. In general, initializing the population with full trees appears to be an obstacle to optimization. Also, multioperation when applied with the init_0 scheme appears to be detrimental.

The theoretical bounds are confirmed by the experiments, suggesting that they might be tight.

7.3 Experimental Analysis of the SMO-GP Variants

We now analyze the experimental results of the SMO-GP variants. We focus on the maximum tree size and population size during execution, and on the expected optimization time.

Table 4 shows two measurements. We list the maximum tree sizes and maximum population sizes that were observed up to the two different but connected events: (1) until the individual X_{opt} with optimal fitness is found, and (2) until the entire true Pareto front P_{Pareto} is represented by the population.

7.3.1 Tree Size

With respect to maximum tree size we can note that with both init_0 and init_n initialization schemes and both single- and multioperation variants the tree size is always very close to the theoretical minimum of $2n - 1$ except for INV, in which there is an increase of about 6% in single operation mode and about 30%–37% in multioperation mode. The interquartile range in these data is minimal, often zero in single operation mode and up to 5% in multioperation mode. It is worth observing that the maximum attained tree size is quite independent of the initialization scheme.

7.3.2 Population Size

While $T_{\max}$ already appears as a factor in the computational complexity bounds for the (1+1)-GP variants, the impact of $P_{\max}$ on SMO-GP is not yet completely clear. It is reasonable to presume that for large populations, for instance, exponential in n, the expected optimization time grows because of the lower probability of selecting the correct individual to improve. Unfortunately it is not clear how often such a large population occurs, since this depends on factors such as the number of different objectives and the fitness levels for each of these objectives.

For the sorting problem, four out of five of the considered sortedness measures yield a linear number of trade-offs, hence population individuals, between fitness value and complexity. Only one of the fitness functions, namely INV, can potentially generate a quadratic number of trade-offs. However, our experiments showed that even in the case of INV the maximum population size is mostly *about* n and always linear in n (see Figure 3, where the population size has been divided by $\log n$ and n). As for the maximum population size, there is no evident correlation between the choice of a particular initialization scheme and the maximum population size.

7.3.3 Average Case Optimization Time

Figure 4 shows the distribution of the expected optimization time for SMO-GP. For multiobjective algorithms the expected optimization time is the number of evaluations

Table 4: The maximum tree sizes and the maximum population sizes encountered for SMO-GP on the multiobjective problem variants: (1) until the individual X_{opt} with maximum fitness is found, (2) until the entire true Pareto front P_{Pareto} is represented by the population. Shown are the median m and interquartile ranges iqr.

		F(X)	n	Maximum Tree Size				Maximum Population Size			
				to X_{opt}		to P_{Pareto}		to X_{opt}		to P_{Pareto}	
				m	iqr	m	iqr	m	iqr	m	iqr
SMO-GP, with $k = 1$	init_0	INV	80	169	4	169	4	85	1	85	1
		LAS	80	159	0	159	0	81	0	81	0
		HAM	80	159	0	159	0	81	0	81	0
		EXC	80	159	2	159	2	81	1	81	1
		RUN	80	159	0	159	0	81	0	81	0
	init_n	INV	80	173	6	173	6	86	2	86	2
		LAS	80	159	0	159	0	81	0	81	0
		HAM	80	159	0	159	0	81	0	81	0
		EXC	80	159	2	159	2	81	1	81	1
		RUN	80	159	2	159	2	81	0	81	0
SMO-GP, with $k = 1 + Pois(1)$	init_0	INV	80	183	8	183	8	89	2	89	2
		LAS	80	159	2	159	2	81	0	81	0
		HAM	80	159	2	159	2	81	0	81	0
		EXC	80	161	2	161	2	81	1	81	1
		RUN	80	161	2	161	2	81	0	81	0
	init_n	INV	80	185	10	185	10	89	3	89	3
		LAS	80	159	2	159	2	81	0	81	0
		HAM	80	159	2	159	2	81	0	81	0
		EXC	80	161	0	161	0	81.5	1	81.5	1
		RUN	80	161	2	161	2	81	0	81	0

to reach the true Pareto front. However, since for our experiments these two measures almost always coincided, we droped the latter and promoted the comparison between (1+1)-GP variants and SMO-GP variants.

As can be seen from the plots, the theoretical bounds on HAM, EXC, and RUN are always verified, suggesting they are tight. As for INV and LAS, the polynomial lines show a strong indication toward a runtime in $\Omega(n^3 \log n)$, mostly close to $O(n^3 \log n)$. Overall, when n is large, the single operation mode seems to yield better factors for the polynomials and a lower failure rate than the multioperation mode.

8 Summary

Existing computational complexity analyses of simple genetic programming has resulted in many insights into the inner workings. Through our theoretical and experimental investigations, we contribute to the understanding of the algorithms.

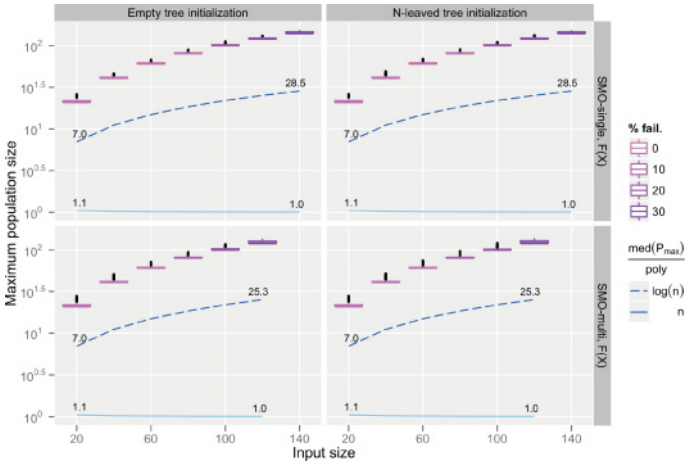

Figure 3: Maximum population size for INV in SMO-GP, possibly quadratic but practically linear in n.

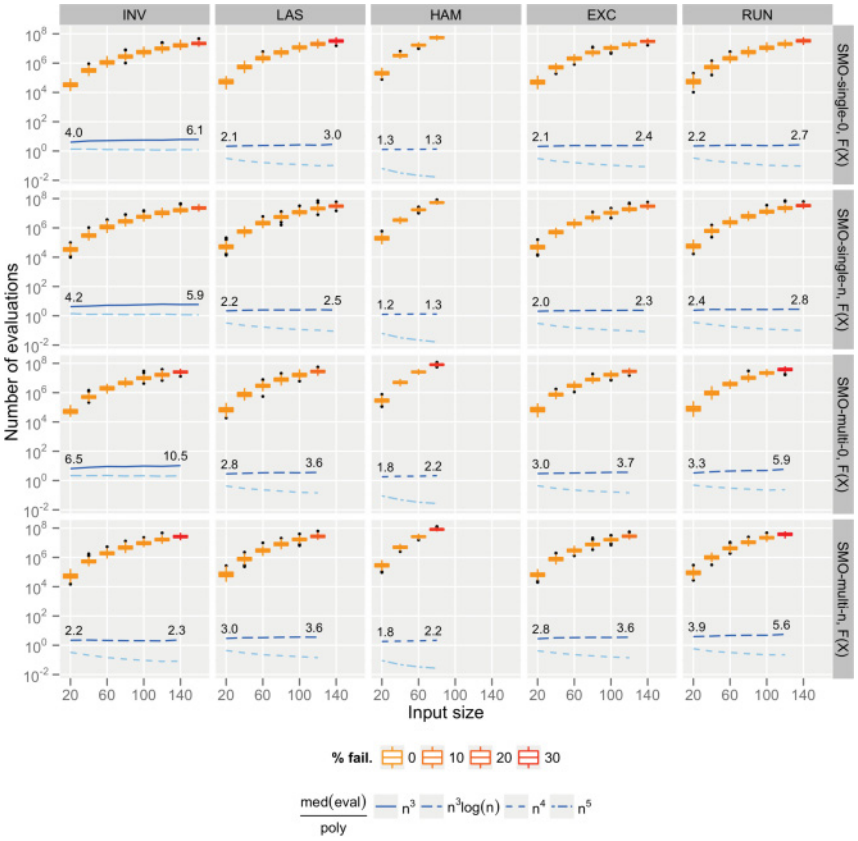

Figure 4: Box plots showing the number of evaluations required until the first individual with optimal fitness was found. This multiobjective approach is more reliable (albeit slower) for solving the problem than the (1+1)-GP setups of Figures 1 and 2.

Table 5: Single-objective problems: summary of proven bounds from Table 1 and our average case conjectures.

F(X)	(1+1)-GP*, F(X)		(1+1)-GP, F(X)	
	single	multi	single	multi
INV	$O(n^3 T_{\max})$	$O(n^3 T_{\max})$	$O(n \log n T_{\max})$ †	$O(n \log n T_{\max})$ †
LAS	∞	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$	$O(n \log n T_{\max})$ †	$O(n \log n T_{\max})$ †
HAM	∞	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$	$O(n^3 T_{\max})$ †	$O(n^3 T_{\max})$ †
EXC	∞	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$	$O(n^3 T_{\max})$ †	$O(n^3 T_{\max})$ †
RUN	∞	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$ †	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$ †

† indicates case conjectures, T_{init} denotes the size of the initial tree, and $T_{\max}$ denotes the size of the largest tree encountered during the optimization.

Table 6: Multiobjective problems: summary of proven bounds from Table 2 and our average case conjectures.

F(X)	(1+1)-GP, MO-F(X)		SMO-GP, MO-F(X)
	single	multi	single/multi
INV	$O(T_{\text{init}} + n^5)$, $O(nT_{\text{init}} + n^3)$ †	$O(nT_{\text{init}} + n^3)$ †	$O\left(n^2 T_{\text{init}} + n^5\right)$, $O(nT_{\text{init}} + n^3 \log n)$ †
LAS	$O(T_{\text{init}} + n^2 \log n)$	$O(T_{\text{init}} + n^2 \log n)$	$O(nT_{\text{init}} + n^3 \log n)$ †
HAM	∞, $O(n^3)$ †	$O(n^3)$ †	$O(nT_{\text{init}} + n^4)$
EXC	∞	$O(nT_{\text{init}} + n^5)$ ‡	$O(nT_{\text{init}} + n^3 \log n)$
RUN	∞	$\Omega\left(\left(\frac{n}{e}\right)^n\right)$ †	$O(nT_{\text{init}} + n^3 \log n)$

† indicates average case conjectures.
‡ marks a conjecture based on the idea that a single *exchange* operation can be simulated with HVL-Prime in time $O(n^4)$. T_{init} denotes the size of the initial tree, and $T_{\max}$ denotes the size of the largest tree encountered during the optimization.

We discussed two methods for dealing with bloat that frequently occurs when using such a representation. In order to point out the differences between these two approaches, we examined different measures of sortedness that have been analyzed for evolutionary algorithms with fixed-length representations. Interestingly, our analysis for the parsimony approach shows that variable-length representations might have difficulties when dealing with simple measures of sortedness because of the presence of local optima. Contrary to this, our runtime analysis for simple multiobjective algorithms shows that they compute the whole Pareto front for all examined sortedness measures in expected polynomial time. In order to complement the theoretical results, we carried out comprehensive experimental investigations.

Crucial parameters in the theoretical analyses are the size of the largest solution encountered during the run of the algorithm, as well as the population size when dealing with multiobjective approaches. In addition, just a few runtime bounds for the multioperation variants are known so far, and the tightness of all bounds is unclear.

Our empirical investigations allow us to conjecture average case complexities where our theoretical analyses left gaps (see Tables 5 and 6):

- **(1+1)-GP, F(X)**: When no bloat control is applied, the algorithm fails regularly to solve RUN. INV and LAS appear to be solvable in $O(n^2 \log n)$, while EXC and HAM are solvable in $O(n^4)$.

- **(1+1)-GP*, F(X)**: This situation changes quite dramatically for the worse when introducing the minimal bloat control mechanism of accepting new solutions only if they are of better fitness. INV is solved in $O(n^4)$ (as theory predicted), assuming a maximum tree size $T_{\max} = O(n)$ (see Table 3). All other sortedness measures are unsuccessful when the initial tree already has n leaves. When initializing with the empty tree, the single-mutation variant achieves a runtime of $O(n^2 \log n)$ on LAS, EXC, and RUN, and a runtime of $O(n^3)$ on HAM.
- **(1+1)-GP, MO-F(X)**: Here, the combination of applying just a single mutation at a time and initializing with the empty tree is the most successful one. When initialized with trees with $C(X) = 2n - 1$, the algorithm has some chance to get stuck in a local optimum on MO-HAM but still achieves an upper bound of $O(n^3)$ in the average case.
- **SMO-GP, MO-F(X)**: All problems are solved in $O(n^3 \log n)$, except for MO-HAM, which is solved on average in $O(n^4)$. Regarding the missing proofs, it should now be easy to show the $O(n^3 \log n)$ for MO-INV, assuming that the maximum population size $P_{\max} = O(n)$, as supported by Figure 3.

Note that our results are based on an initial tree size T_{init}, which is always linear in n. Consequently, the $O(n \log n)$ term always dominates the T_{init} term suggested by theoretical results. In spite of this, it is easy to prove that the T_{init} term can become relevant when arbitrarily large initial trees are used.

To continue this avenue of research, it would be interesting to theoretically prove the conjectured bounds, and to investigate how the maximum tree sizes and population sizes can be bounded in different scenarios.

In general, in order to narrow the gap between theory and application, the investigated problems need to resemble real-world problems more closely. One direction that we might take is the analysis of variable-length algorithms when they are used for symbolic regression, which is one of the most relevant use cases for genetic programming.

References

Auger, A., and Doerr, B. (2011). *Theory of randomized search heuristics: Foundations and recent developments*. Singapore: World Scientific Publishing.

Briest, P., Brockhoff, D., Degener, B., Englert, M., Gunia, C., Heering, O., Jansen, T., Leifhelm, M., Plociennik, K., Röglin, H., Schweer, A., Sudholt, D., Tannenbaum, S., and Wegener, I. (2004). Experimental supplements to the theoretical analysis of EAs on problems from combinatorial optimization. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pp. 21–30. Lecture Notes in Computer Science, Vol. 3242.

Cathabard, S., Lehre, P. K., and Yao, X. (2011). Non-uniform mutation rates for problems with unknown solution lengths. In *Proceedings of the Workshop on Foundations of Genetic Algorithms*, pp. 173–180.

Doerr, B., and Goldberg, L. A. (2010). Drift analysis with tail bounds. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pp. 174–183. Lecture Notes in Computer Science, Vol. 6238.

Doerr, B., and Happ, E. (2008). Directed trees: A powerful representation for sorting and ordering problems. In *Proceedings of the IEEE World Congress on Computational Intelligence*, pp. 3606–3613.

Durrett, G., Neumann, F., and O'Reilly, U.-M. (2011). Computational complexity analysis of simple genetic programing on two problems modeling isolated program semantics. In *Proceedings of the Workshop on Foundations of Genetic Algorithms*, pp. 69–80.

Evolved Analytics LLC (2010). *DataModeler 8.0*. www.evolved-analytics.com

Falco, I. D., Tarantino, E., Cioppa, A. D., and Gagliardi, F. (2005). A new variable-length genome genetic algorithm for data clustering in semiotics. In *Proceedings of the ACM Symposium on Applied Computing*, pp. 923–927.

Friedrich, T., He, J., Hebbinghaus, N., Neumann, F., and Witt, C. (2010). Approximating covering problems by randomized search heuristics using multi-objective models. *Evolutionary Computation*, 18:617–633.

Giel, O. (2003). Expected runtimes of a simple multi-objective evolutionary algorithm. In *Proceedings of the Congress on Evolutionary Computation*, pp. 1918–1925.

Giel, O., and Lehre, P. K. (2010). On the effect of populations in evolutionary multi-objective optimisation. *Evolutionary Computation*, 18:335–356.

Kötzing, T., Sutton, A. M., Neumann, F., and O'Reilly, U.-M. (2012). The max problem revisited: The importance of mutation in genetic programming. In *Proceedings of the International Conference on Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1333–1340.

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*. Cambridg, MA: MIT Press.

Lässig, J., and Sudholt, D. (2010). Experimental supplements to the theoretical analysis of migration in the island model. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pp. 224–233. Lecture Notes in Computer Science, Vol. 6238.

Laumanns, M., Thiele, L., and Zitzler, E. (2004). Running time analysis of multiobjective evolutionary algorithms on pseudo-Boolean functions. *IEEE Transactions on Evolutionary Computation*, 8:170–182.

Lee, C., and Antonsson, E. K. (2000). Variable length genomes for evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, p. 806.

Mambrini, A., Manzoni, L., and Moraglio, A. (2013). Theory-laden design of mutation-based geometric semantic genetic programming for learning classification trees. In *Proceedings of the Congress on Evolutionary Computation*, pp. 416–423.

Moraglio, A., Mambrini, A., and Manzoni, L. (2013). Runtime analysis of mutation-based geometric semantic genetic programming on Boolean functions. In *Proceedings of the Conference on Foundations of Genetic Algorithms*, pp. 119–132.

Neumann, F. (2012). Computational complexity analysis of multi-objective genetic programming. In *Proceedings of the International Conference on Genetic and Evolutionary Computation (GECCO)*, pp. 799–806.

Neumann, F., and Wegener, I. (2005). Minimum spanning trees made easier via multi-objective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 763–770.

Neumann, F., and Witt, C. (2010). *Bioinspired computation in combinatorial optimization: Algorithms and their computational complexity*. New York: Springer.

Neumann, F., O'Reilly, U.-M., and Wagner, M. (2011). Computational complexity analysis of genetic programming: Initial results and future directions. In R. Riolo, E. Vladislavleva, and J. H. Moore (Eds.), *Genetic programming theory and practice IX*, *Genetic and evolutionary computation*, pp. 113–128. New York: Springer.

Nguyen, A., Urli, T., and Wagner, M. (2013). Single- and multi-objective genetic programming: New bounds for weighted order and majority. In *Proceedings of the Workshop on Foundations of Genetic Algorithms*, pp. 161–172.

O'Reilly, U.-M. (1995). An analysis of genetic programming. Unpublished doctoral dissertation, Carleton University, Ottawa, ON K1S 5B6.

O'Reilly, U.-M., and Oppacher, F. (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pp. 397–406. Lecture Notes in Computer Science, Vol. 866.

Scharnow, J., Tinnefeld, K., and Wegener, I. (2004). The analysis of evolutionary algorithms on sorting and shortest paths problems. *Journal of Mathematical Modelling and Algorithms*, 3:349–366.

Urli, T., Wagner, M., and Neumann, F. (2012). Experimental supplements to the computational complexity analysis of genetic programming for problems modelling isolated program semantics. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pp. 102–112. Lecture Notes in Computer Science, Vol. 7491.

Wagner, M., and Neumann, F. (2012). Parsimony pressure versus multi-objective optimization for variable length representations. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pp. 133–142. Lecture Notes in Computer Science, Vol. 7491.

Wagner, M., and Neumann, F. (2014). Single- and multi-objective genetic programming: New runtime results for sorting. In *Proceedings of the IEEE Congress on Evolutionary Computation, Special Session*, pp. 125–132.

Wegener, I. (2002). Methods for the analysis of evolutionary algorithms on pseudo-Boolean functions. In R. Sanker, M. Mohammadian, and X. Yao (Eds.), *Evolutionary optimization*, pp. 349–369. New York: Springer.