2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Single- and Multi-Objective Genetic Programming: New Runtime Results for SORTING

Markus Wagner and Frank Neumann

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 optimisation process. Consequently, problems that are relatively easy to optimise can not be handled by variable-length evolutionary algorithms.**

In this article, we present several new bounds for different single- and multi-objective algorithms on the sorting problem, a problem that typically lacks independent and additive fitness structures.

I. Introduction

Genetic programming (GP) [11] is the most prominent example of a variable-length evolutionary algorithm, as it often evolves tree-like solutions for a given problem. Just recently, the first computational complexity results on this type of algorithm have been obtained, following the line of successful research on evolutionary algorithms with fixed-length representation (see the books by Auger and Doerr [1], Neumann and Witt [16] for an overview). In general, variable-length representations increase the search space significantly, and it is desirable to better understand the behaviour of algorithms using such representations from a theoretical point of view.

For example, Cathabard, Lehre, and Yao [2] investigated non-uniform mutation rates for problems with unknown solution lengths. A simple evolutionary algorithm was used to find a bitstring with an unknown number of leading ones and although the bitstring 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, Neumann, and O'Reilly [5]. The tackled problems were separable, with independent and additive fitness structures. Similarly, Kötzing, Sutton, Neumann, and O'Reilly [10] analysed simple GP algorithms for the MAX problem. Recently, a new form of GP called Geometric Semantic Genetic Programming was investigated, with positive results for general classes of traditional GP problems (see, e.g., [13]).

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 common way of dealing with the bloat problem is inspired by Occam's Razor: in the case that two solutions are equal in quality, the solution of lower complexity shall be preferred. Another common way of coping with the bloat problem, is to use a multi-objective approach that uses a population representing the different trade-offs according to the original goal function and the complexity of a solution. This approach can be found even in industrially used GP packages such as Datamodeller [6]. Both approaches of coping with the bloat problem have recently been examined for different problems in the context of genetic programming [14, 18, 23].

In this article, we will investigate the sorting problem, which is one of the most basic problems in computer science. It is also the first combinatorial optimisation problem for which computational complexity results have been obtained in the area of discrete evolutionary algorithms [4, 21]. In [21], sorting is treated as an optimisation problem where the task is to minimise the unsortness of a given permutation of the input elements. To measure unsortness, different fitness functions have been introduced in the past and studied with respect to the difficulty of being optimised by permutation-based evolutionary algorithms. Depending on the chosen measure and in contrast to WORDER and WMAJORITY, the sorting problem can typically not be split into subproblems that can be solved independently. Consequently, the dependencies between the subproblems can have a significant impact on the time needed to solve the overall problem.

With our work, we continue the analyses started in [23], which focussed on the advantages of a parsimonious algorithm over a multi-objective one. Here, we present several new bounds for a total of three single- and multi-objective algorithms using five sortedness unmeasurements.

This article is organised as follows. We first introduce the sorting problem in Section II. Afterwards, we present the different genetic programming algorithms that we will analyse in Section III. Then, in Section IV we investigate the single-objective approach, in Section V the parsimony approach, and in Section VI the multi-objective approach. Our findings are summarised in the concluding Section VII.

II. Preliminaries

Our goal is to investigate theoretically the differences between bloat-control mechanisms for genetic programming. In our investigations, we will treat the algorithms and problems analysed in [5, 14, 17, 18, 23]. We consider tree-based genetic programming, where a possible solution is represented by a syntax tree. The inner nodes of such a tree are labelled by function symbols from a set F and the leaves of the tree are labelled 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

Frank Neumann and Markus Wagner are with the Optimisation and Logistics Group, School of Computer Science, The University of Adelaide (email: {frank.neumann, markus.wagner}@adelaide.edu.au).

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

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

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

single binary function "join" J. Effectively, we use J's 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 the computational complexity analysis of evolutionary algorithms with fixed-length representations, namely the sorting problem (SORTING). Scharnow et al. [21] considered SORTING as an optimisation problem, where different fitness functions measure the sortedness of a permutation of elements. It was discovered that different fitness functions lead to problems of different difficulties.

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

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

holds, where $<$ is the order on T. Without loss of generality, we assume $\pi_{opt} = id$, i. e. $\pi_{opt}(i) = i$ 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. [21] for the analysis of permutation-based evolutionary algorithms. The authors of that article investigate SORTING as an optimisation problem where the goal is to maximise the sortedness of a given permutation. We will consider the following fitness functions measuring the sortedness of a given permutation introduced in [21]:

- INV(π), measuring the number of pairs in correct order (larger values are better),
- HAM(π), measuring the number of elements at correct position, which is the number of indices i such that $\pi(i) = i$ (larger values are better),
- RUN(π), measuring the number of maximal sorted blocks, which is the number of indices i such that $\pi(i+1) < \pi(i)$ plus one (smaller values are better),
- LAS(π), measuring the length of the longest ascending subsequence (larger values are better),
- EXC(π), measuring the minimal 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 inorder 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 that $EXC(\pi)$ can be computed in linear time, based on the cycle structure of permutations. If the sequence is sorted, 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, $EXC(\pi) = k$.

We will 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 will use the sortedness measures as above 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.

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) $= 1 + 1 = 2$, and EXC(X) $= 4 + 1 + 1 = 6$.

MO-INV, MO-HAM, MO-RUN, MO-LAS and MO-EXC are variants of the above-described problems. They take the complexity C of a syntax tree (computed by the number of leaves of the tree) as the second objective, e. g., MO-INV (X) = (INV (X), C(X)). Optimisation algorithms can then use this to cope with the bloat problem: if two solutions have the same fitness value, then the solution of lower complexity can be preferred.

III. Algorithms

All GP algorithms that we analyse in this article only use the mutation operator HVL-Prime to generate offspring. HVL-Prime is an update of O'Reilly's HVL mutation operator [19, 20]. 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 step of the algorithms, k mutations are applied to the selected solution. For the single-operation variants of the algorithms, $k = 1$ holds. For the multi-operation variants, the number of operations performed is drawn each time from the distribution $k = 1 + Pois(1)$, where $Pois(1)$ is the Poisson distribution with parameter 1.

The algorithm (1+1)-GP* that we investigate first has no explicit mechanism to control bloat whatsoever. The only

Algorithm 2: HVL-Prime mutation operator

Mutate Y by applying HVL-Prime k times: each time randomly choose either insert, substitute or delete.
if *Insert* **then**
    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.
if *Substitute* **then**
    Replace a randomly chosen leaf $v \in Y$ by a randomly chosen leaf $u \in L$.
if *Delete* **then**
    Choose a leaf node $v \in Y$ randomly with parent p and sibling u. Replace p by u and delete p and u.

Algorithm 3: (1+1)-GP*-single for maximisation

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 $X := Y$;

feature that can potentially prevent the solution's size to become 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 and by the number of possible fitness improvements that can be performed.[1]

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 neighbourhood. In the case of maximisation, Y replaces X if $f(Y) > f(X)$ holds. Minimisation problems are tackled in the analogous way.

The single-objective variant called (1+1)-GP is identical to the just described (1+1)-GP*, with the exception that, in the case of maximisation, Y replaces X if $f(Y) \geq f(X)$ holds. Again, minimisation 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 truly 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 multi-objective variants of the presented sortedness measures, e. g., MO-INV. Without loss of generality, we assume that C is to be

[1]Note that 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 4: SMO-GP

Choose an initial solution X;
Set $P := \{X\}$;
repeat
    Choose $X \in P$ uniformly at random;
    Set $Y := X$;
    Apply mutation to Y;
    if $\{Z \in P \mid Z \succeq Y\} = \emptyset$ **then** set $P := (P \setminus \{Z \in P \mid Z \succ Y\}) \cup \{Y\}$;

minimised and all fitness functions F, except RUN and EXC, are maximised. In the parsimony approach, we optimise the above-defined multi-criteria fitness functions MO-F$(X) = (F(X), C(X))$ with respect to the lexicographic order, that is, MO-F$(X) \geq$ MO-F(Y) holds 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 (SMO-GP, see Algorithm 4) algorithm introduced by Neumann [14] and motivated by the SEMO algorithm for fixed-length representations by Laumanns, Thiele, and Zitzler [12]. Variants of SEMO have been frequently used in the runtime analysis of evolutionary multi-objective optimisation for fixed length representations [see 7–9, 15, 16].

In this multi-objective variable-length algorithm, we treat the two criteria F and C as equally important. In order to compare two solutions, we consider the classical Pareto dominance relations:

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

A *Pareto optimal solution* is a solution that is not dominated by any other solution in the search space. All Pareto optimal solutions together form the Pareto optimal set, and the set of corresponding objective vectors forms the Pareto front. The classical goal in multi-objective optimisation is to compute for each objective vector of the Pareto front a Pareto optimal solution. Or, if the Pareto front is too large, the goal then is to find a representative subset of the front, where the definition of 'representative' depends on the choice of the conductor.

SMO-GP is a population-based approach that starts with a single solution and it maintains a set of non-dominated solutions obtained during the optimisation run. This set of solutions constantly approximates the true Pareto front, i.e. 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 that differs from SMO-GP-single by choosing k according to $1 + Pois(1)$.

IV. Standard Approach Without Bloat-Control

The algorithm (1+1)-GP* (see Algorithm 3) that we investigate first has no mechanism to control bloat whatsoever. The only feature that can potentially prevent the solution's size to become 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 and by the number of possible fitness improvements that can be performed.

A. Upper Bound

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

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 has originally been introduced for the analysis of elitist evolutionary algorithms (see, e. g., [24]), where the fitness of the current search point can never decrease. Although the used 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 sub-operations insertion and substitution. In combination with fitness levels defined individually for the used sortedness measures, this gives us the runtime bounds in this section.

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

Theorem 1. *The expected optimisation time is $O(n^3 T_{max})$ for the (1+1)-GP*-single and (1+1)-GP*-multi, using the sortedness measure INV.*

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

$$A_i = \{\pi \,|INV(\pi) = i\}.$$

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.[2] For this particular insertion, we consider the lexicographically smallest pair (i, j), $i < j$, that is currently incorrect: putting i directly before j makes this pair correct. We now have to show that this does not make any other pair which was previously correct, incorrect. Assume there is a pair (k, l), $k < l$ that was previously correct and has become incorrect due to 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 (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 introduced inner J-node is at least $\frac{1}{2}$. This, together with the probability of selecting the right element to add, which is bound by $\frac{1}{n}$, and the probability of adding it to the right position in the tree, which is bound by $\frac{1}{T_{max}}$, gives us a lower bound on the probability for doing such an improvement in (1+1)-GP*-single[3]

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

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

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

□

B. 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 during the optimisation process. This shows that evolving a solution with this GP system is much harder than working with the permutation-based EA presented in [21], where only the sortedness measure RUN leads to an exponential optimisation 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 inorder parse of the tree. We restrict ourselves to the case where we initialise with a tree of size linear in n and show that even this leads to difficulties for the mentioned sortedness measures. Note, that a linear size 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.

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

[3]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 it's right sibling.

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

Proof. We consider (1+1)-GP*-single first. It is clear that with a single HVL-Prime application, only one of the leftmost ns can be removed. For an improvement in the sortedness based on RUN or LAS, all leftmost $n+1$ leaves have to be removed at once. Obviously, this cannot be done by the (1+1)-GP*-single, resulting in an infinite runtime.

(1+1)-GP*-multi can only improve the fitness by removing the leftmost $n+1$ leaves. Hence, in order to successfully improve the fitness, at least $n+1$ sub-operations have to be performed, assuming that we, in each case, delete one of the leftmost ns. Because the number of sub-operations per mutation is distributed as $1 + Pois(1)$, the Poisson random variable has to take a value of at least n. This implies that the probability for such a step is $e^{-\Omega(n)}$ and the expected waiting time for such a step is therefore $e^{\Omega(n)}$. □

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 this 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 optimisation time of (1+1)-GP*-single and (1+1)-GP*-multi is infinite and $e^{\Omega(n)}$, respectively.*

Proof. We use similar ideas as in the previous proof. Again, it is not possible for (1+1)-GP*-single to improve the fitness in a single step, as all $n+1$ leftmost leaves have to be removed in order for the rightmost n to become expressed. Additionally, a leaf labelled 1 has to be inserted at the beginning, or alternatively, one of the $n+1$ leaves labelled n has to be replaced by a 1. This results in a minimum number of $n+1$ sub-operations that have to be performed by a single HVL-Prime application, leading to the lower bound of $e^{\Omega(n)}$ for (1+1)-GP*-multi. □

V. 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 maximisation, Y replaces X if $f(Y) \geq f(X)$.

In [23] it was shown that the optimisation time of (1+1)-GP-single on MO-EXC, MO-RUN and MO-HAM is infinite, when initialised 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 optimisation time on MO-LAS is as follows. Given a tree T with its tree size of T_{init}, and its sortedness $LAS(T) = k < n$. 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} from Section IV-B, which have previously been problematic.

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

Proof. For the analysis, we consider two phases. First, we show that we arrive at a tree with fitness k and k leaves after $O\left(T_{init} + n \log n\right)$ steps. Afterwards, we analyse the time needed to get from there to the optimal solution.

1. Phase. Initially, let $LAS(T) = k$ be the fitness of the current tree T with s leaves. Then, the 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 can be lower bounded by

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

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 as for d to increase, k would have to decrease, which is impossible, as the primary objective is the maximisation 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, we can bound the expected runtime for this first phase:

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

2. Phase. Next, we investigate the time needed in the second phase to arrive at the optimum. Therefore, we again apply the above-described fitness-based partitions method. We define the fitness levels $A_1, \ldots, A_n$ with $A_i = \{T \,|\, LAS(T) = i\}$. 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 $\frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{n} \cdot \frac{n-k}{n} = \frac{1}{6} \cdot \frac{n-k}{n^2}$. Thus, 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 optimisation time of the algorithm is upper bounded by $O\left(T_{init} + n^2 \log n\right)$. □

Theorem 5. *The expected optimisation time of (1+1)-GP-single on MO-INV is* $O\left(T_{init} + n^5\right)$.

Proof. For our analysis, we draw upon results from the Theorems 1 and 4. First, after $O\left(n \log n + T_{init}\right)$ steps, we arrive at a non-redundant 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 optimisation time by

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

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

Therefore, we take an alternate approach, by looking at a sequence of steps $t = poly(n)$. Let T_{init} to be a tree with size $|T_{init}| = poly(n)$. 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 $T_{max} = T_{init} + n^{1+\epsilon}$ within $t = poly(n)$ time steps, with high probability.

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

Proof. We will 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 non-redundant tree after the m-th fitness-increasing insertion. In the following, let i be the number of redundant elements in the tree, and let j be the number of non-redundant 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 optimisation 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[4] (with $\delta = 1$) we get that

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

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 [3]).

In our case, $\Phi(x) = i$ 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 non-optimal solutions, and $E[\Phi(x_{new})] \leq \left(i - \frac{i}{6en}\right) = \left(1 - \frac{1}{\nu(n)}\right)\Phi(x)$. 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 will again apply the multiplicative drift with tail bounds, as used above. 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 $E[\Phi(x_{new})] \leq \left(j - \frac{j}{6en^2}\right) = \left(1 - \frac{1}{\nu(n)}\right)\Phi(x)$ we get, $\nu(n) = 6en^2$ and $\delta = 1$. Consequently, by applying Theorem 1 from [3]), 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 $1 - n^{-c} = 1 - o(1)$. □

[4]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: $P(X \geq (1+\delta)E[X]) \leq e^{-E[X]\delta^2/3}, 0 < \delta \leq 1$

VI. Multi-Objective Approach

In this section, we consider the Simple Evolutionary Multi-Objective Genetic Programming (SMO-GP, see Algorithm 4) algorithm introduced by Neumann [14] and motivated by the SEMO algorithm for fixed length representations by Laumanns et al. [12]. We analyse the expected number of iterations before the set of non-dominated solutions becomes the true Pareto front. We call this the *expected optimisation time* of SMO-GP algorithms.

The following lemma bounds the expected time until the empty solution has been included into the population, when considering an arbitrary optimisation problem:

Lemma 1 (Neumann [14])**.** *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 $O(kI_{init})$.*

Theorem 7. *The expected optimisation time of SMO-GP-single and SMO-GP-multi is $O(n^2 I_{init} + n^5)$ on MO-INV, and $O(nI_{init} + n^3 \log n)$ 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 $O\left(n^2 I_{init}\right)$ steps. First, note that each Pareto optimal solution with complexity $2i-1$ has an INV-value of $\sum_1^{i-1} i$, if $i \geq 2$.[5]

Second, as above, we will bound the time needed to discover the whole Pareto front, once the empty solution is introduced into the population. Let us 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 optimisation time once we can bound the probability for such steps to happen. Choosing X for mutation has probability at least $\frac{1}{n(n-1)/2+1}$ as the population size is upper bound by $n(n-1)/2+1$. Next, the mutation step carrying out just one operation happens with 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 can be bounded by $1/n$. However, the correct position for such a randomly chosen element has to be chosen, in order to produce the Pareto optimal solution 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 $\frac{1}{n(n-1)/2+1} \cdot \frac{1}{3e} \cdot \frac{1}{2n} \cdot \frac{1}{n}$.

Thus, as there are only n Pareto-optimal improvements possible once the empty solution is introduced into the population, the expected time until all Pareto optimal solutions have been generated is:

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

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, as only n different LAS-values are possible, the empty solution is produced after an expected number of $O(nI_{init})$ steps. Just as above, let us 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 inserting any of the missing $n-i$ elements into its correct position in the Pareto optimal individual X with $\text{LAS}(X) = C(X) = 2i-1$.

Thus, as there are only n Pareto-optimal improvements possible once the empty solution is introduced into the population, the expected time until all Pareto optimal solutions have been generated is:

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

VII. Conclusions

With this article, we contribute to the understanding of variable-length algorithms with our theoretical investigations. We show that parsimonious and multi-objective approaches can help algorithms to solve problems that are otherwise unsolvable.

Tables I and II summarise our theoretical findings, list existing bounds, and show open problems. As can be observed from the tables, all bounds take into account tree sizes of some kind: either the maximum solution size T_{max}, 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. A comprehensive experimental investigation on the tightness of the presented bounds (similar to the one done in [22]) will be included in the journal version of this article.

In order to narrow the gap between theory and application over the next couple of years, 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* uses of genetic programming.

[5]For the sake of readability, we will omit in the following the special cases for $i = 0$ and $i = 1$.

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

TABLE I

SUMMARY OF COMPUTATIONAL COMPLEXITY BOUNDS FOR SINGLE-OBJECTIVE VARIANTS. THE QUESTION MARK INDICATES COMBINATIONS FOR WHICH WE DO NOT KNOW ANY BOUNDS. *INDICATES NEW BOUNDS PRESENTED IN THIS ARTICLE.

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

TABLE II

SUMMARY OF COMPUTATIONAL COMPLEXITY BOUNDS FOR MULTI-OBJECTIVE VARIANTS. † INDICATES A BOUND THAT HOLDS WITH PROBABILITY $1 - o(1)$. QUESTION MARKS INDICATE COMBINATIONS FOR WHICH WE DO NOT KNOW ANY BOUNDS. *INDICATES NEW BOUNDS PRESENTED IN THIS ARTICLE.

REFERENCES

[1] A. Auger and B. Doerr. *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 *FOGA*, pp. 173–180. ACM, 2011.

[3] B. Doerr and L. A. Goldberg. Drift analysis with tail bounds. In *PPSN*, pp. 174–183. Springer, 2010.

[4] B. Doerr and E. Happ. Directed trees: A powerful representation for sorting and ordering problems. In *CEC*, pp. 3606–3613. IEEE, 2008.

[5] G. Durrett, F. Neumann, and U.-M. O'Reilly. Computational complexity analysis of simple genetic programing on two problems modeling isolated program semantics. In *FOGA*, pp. 69–80. ACM, 2011.

[6] Evolved Analytics LLC. *DataModeler 8.0*. Evolved Analytics LLC, 2010.

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

[8] O. Giel. Expected runtimes of a simple multi-objective evolutionary algorithm. In *CEC*, pp. 1918–1925, 2003.

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

[10] T. Kötzing, A. M. Sutton, F. Neumann, and U.-M. O'Reilly. The max problem revisited: the importance of mutation in genetic programming. In *GECCO*, pp. 1333–1340. ACM, 2012.

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

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

[13] A. Moraglio, A. Mambrini, and L. Manzoni. Runtime analysis of mutation-based geometric semantic genetic programming on boolean functions. In *FOGA*, pp. 119–132. ACM, 2013.

[14] F. Neumann. Computational complexity analysis of multi-objective genetic programming. In *GECCO*, pp. 799–806. ACM, 2012.

[15] F. Neumann and I. Wegener. Minimum spanning trees made easier via multi-objective optimization. In *GECCO*, pp. 763–770. ACM, 2005.

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

[17] F. Neumann, U.-M. OReilly, and M. Wagner. Computational complexity analysis of genetic programming - initial results and future directions. In *GPTP*, pp. 113–128. Springer, 2011.

[18] A. Nguyen, T. Urli, and M. Wagner. Single- and multi-objective genetic programming: new bounds for weighted order and majority. In *FOGA*, pp. 161–172. ACM, 2013.

[19] U.-M. O'Reilly. *An Analysis of Genetic Programming*. PhD thesis, Carleton University, 1995.

[20] U.-M. O'Reilly and F. Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In *PPSN*, pp. 397–406. Springer, 1994.

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

[22] T. Urli, M. Wagner, and F. Neumann. Experimental supplements to the computational complexity analysis of genetic programming for problems modelling isolated program semantics. In *PPSN*, pp. 102–112. Springer, 2012.

[23] M. Wagner and F. Neumann. Parsimony pressure versus multi-objective optimization for variable length representations. In *PPSN*, pp. 133–142. Springer, 2012.

[24] I. Wegener. Methods for the analysis of evolutionary algorithms on pseudo-Boolean functions. In *Evolutionary Optimization*, pp. 349–369. Kluwer, 2002.