Contents

1
Computational Complexity Analysis of Genetic Programming 1
Frank Neumann, Una-May O'Reilly and Markus Wagner

Chapter 1

COMPUTATIONAL COMPLEXITY ANALYSIS OF GENETIC PROGRAMMING - INITIAL RESULTS AND FUTURE DIRECTIONS

Frank Neumann[1], Una-May O'Reilly[2] and Markus Wagner[1]
[1]*School of Computer Science, University of Adelaide, Australia;* [2]*CSAIL, MIT, Cambridge, USA.*

Abstract The computational complexity analysis of evolutionary algorithms working on binary strings has significantly increased the rigorous understanding on how these types of algorithm work. Similar results on the computational complexity of genetic programming would fill an important theoretic gap. They would significantly increase the theoretical understanding on how and why genetic programming algorithms work and indicate, in a rigorous manner, how design choices of algorithm components impact its success. We summarize initial computational complexity results for simple tree-based genetic programming and point out directions for future research.

Keywords: genetic programming, computational complexity analysis, theory

1. Introduction

Genetic programming (GP) (Koza, 1992) is a type of evolutionary algorithm that has proven to be very successful in various fields including modeling, financial trading, medical diagnosis, design and bioinformatics (Poli et al., 2008). It differs from other types of evolutionary algorithms by searching for solutions which are *executable*, i.e. that are program-like functions, which can be interpreted or execute when their input variables and formal parameters are bound. To apply variation operators that transform one candidate solution (or program) into another, conventional GP first converts an executable function into its parse tree (a.k.a. abstract syntax tree) representation then randomly modifies the tree in a way that preserves syntactic correctness but varies its size and/or structure. GP contrasts with other evolutionary algorithms, such as genetic algorithms and evolution strategies, because, rather than optimizing a given objective function, the goal in genetic programming is to search and identify one or more programs that exhibit a set of desired functionality. Usually this functionality is described in terms of desired input-output behaviour.

In contrast to numerous successful applications of genetic programming, the theoretical understanding of GP lags far behind its practical success (see (Poli et al., 2010) for an excellent appraisal of state of the art). While a number of theoretical approaches have been pursued and are active (e. g. schema theory, building block analysis, Markov chains, search space, bloat and problem difficulty analysis), a new direction which can provide proofs of convergence and time and space complexity analysis is most welcome. Our aim is to build up such a theory to gain new theoretical insights into the working principles of GP.

Computational complexity of evolutionary algorithms with binary representations

In the field of evolutionary algorithms that operate on binary solution representations, numerous theoretical insights have been gained in the last 15 years by computational complexity analyses. Initial results were obtained on simple pseudo-Boolean functions which point out basic working principles of simple evolutionary algorithms using binary representations (Droste et al., 2002). Results have been derived for a wide range of classical combinatorial optimization problems such as minimum spanning tress, shortest path and different cutting and covering problems (see e. g. (Neumann and Witt, 2010) for an overview). Problem specific algorithms for some of the classical problems enable them to be solved in polynomial time. We can not and do not expect that evolutionary

algorithms to outperform such algorithms. However, studying the same problems and deriving run time bounds for evolutionary algorithms that solve them, allows us to gain a rigorous theoretical understanding of how these evolutionary algorithms work. Further, it yields insights as to how evolutionary algorithms are useful for tackling NP-hard variants of these problems.

To explain the type of results that might arise from computation complexity analysis of GP algorithms, it is helpful to review complexity analysis results for binary evolutionary algorithms. A good counterpart example is the classical minimum spanning tree problem which is one of the first problems where computational complexity results have been obtained for evolutionary algorithms working with binary representation. Runtime in these studies is always measured as the number of constructed solutions until an optimal solution has been found. In GP we would think of this as the number of fitness evaluations. The runtime bound for a simple evolutionary algorithms called (1+1) EA is $O(m^2(\log n + \log w_{\max}))$, where m is the number of edges, n is the number of vertices, and $w_{\max}$ is the largest weight of an edge in the given graph (Neumann and Wegener, 2007). Given evolutionary algorithms do not use global information, it is remarkable that they can provably solve classical combinatorial optimization problems in such a small amount of time. The analysis carried out in (Neumann and Wegener, 2007) shows rigorously that the result arises because several advantageous mutations can often be carried out in each iteration. This, and related, results have later been used to obtain a runtime bound for the NP-hard multi-objective minimum spanning tree problem (Neumann, 2007). In this case, it is shown that multi-objective evolutionary algorithms obtain a 2-approximation for this NP-hard problem in expected polynomial time if the size of the Pareto front is polynomially bounded. Note that this subsequent analysis considers a more complex evolutionary algorithm—one that is population based and using optimizing more than one objective.

Computational Complexity Analysis of GP

Our long term ambition is to develop, from the present simple state of art, an extended suite of computational complexity analyses for a range of GP algorithms solving model problems which represent important and commons aspects of real world GP counterparts. We intend to describe GP algorithms in a strict mathematical sense which will help lead to a rigorous understanding of GP algorithm convergence and time complexity which, in turn, will reveal the complexity implications of a variety of algorithm features (and specific choices) such as selection, variation

and even bloat control. GP complexity analysis will appear similar to binary evolutionary algorithm analysis with respect to analyzing each algorithms in the context of solving a specific problem. In progressive analyses, along a daunting path, the sophistication of the algorithm, can optimistically ratchet upwards, likely in modest steps, from a simple hill climbing algorithm (e.g. (1+1) EA) with simplified operators towards population based versions with more realistic operators. The analysis, in general, however, will be distinct and contrasting primarily because GP algorithms work with executable candidate solutions. For example, conventional, a.k.a. Koza or tree-style, GP's variation operators modify a tree by changing its size and or structure. The variable size and tree-shape representations will challenge current complexity analysis techniques because the tree size changes over time and genetic material that can be added or deleted anywhere in any amount implies complicated consequences on the likelihood of fitness improvement from parent to child. The variation operators, particularly crossover, are more complicated, than their counterparts in evolutionary algorithms working with binary representations because of they are frequently designed to be quite unconstrained in terms of where and how much material is exchanged. New methods will be needed to analyze them.

We would aim to develop these methods and achieve computational complexity results for a wide range of algorithmic design choices and model problems so that practitioners will gain new insights into their working principles. Theoretical insights would ideally be used to develop even more powerful genetic programming techniques. We believe there can be a close relationship between theoretical work which offers rigorous proofs and substantial applied work that leads to new effective GP algorithms. Additionally, insights gained by the computational complexity analysis will help to teach this growing field of research in a much clearer way to undergraduate and postgraduate students.

In the rest of this chapter, we give an overview on some initial results we have derived and point different areas of research that we think are interesting to work on. We start by defining simplified GP algorithms that have been used to start the computational complexity analysis of genetic programming in Section 2. Afterwards, we present some initial results that have been obtained recently. The first results are on two problem modeling isolated program semantics called ORDER and MAJORITY. We present computational complexity results for these functions in Section 3. In Section 4, we investigate results for fitness functions motivated by the classical SORTING problem. For this problem the different variables in a GP program depend on each other which imposes other difficulties than for ORDER and MAJORITY. After

having giving some insights into recent results we outline possible topics for future work which are often motivated by successful research projects on the computational complexity analysis for binary search spaces.

2. Simple Algorithms

To use tree-based genetic programming (Koza, 1992), one must first choose a set of primitives A, which contains a set F of functions and a set L of terminals. Each primitive has explicitly defined semantics; for example, a primitive might represent a Boolean condition, a branching statement such as an IF-THEN-ELSE conditional, the value bound to an input variable, or an arithmetic operation. Functions are parameterized. Terminals are either functions with no parameters, i.e. arity equal to zero, or input variables to the program that serve as actual parameters to the formal parameters of functions.

In our derivations, we assume that a GP program is initialized by its parse tree construction. In general, we start with a root node randomly drawn from A and recursively populate the parameters of each function in the tree with subsequent random samples from A, until the leaves of the tree are all terminals. Functions constitute the internal nodes of the parse tree, and terminals occupy the leaf nodes. The exact properties of the tree generated by this procedure will not figure into the analysis of the algorithm, so we do not discuss them in depth.

HVL-Prime

The HVL-Prime operator is an update of O'Reilly's HVL mutation operator (O'Reilly, 1995; O'Reilly and Oppacher, 1994) and motivated by minimality rather than inspired from a tree-edit distance metric. HVL first selects a node at random in a copy of the current parse tree. Let us term this the `currentNode`. It then, with equiprobability, applies one of three sub-operations: insertion, substitution, or deletion. Insertion takes place above `currentNode`: a randomly drawn function from F becomes the parent of `currentNode` and its additional parameters are set by drawing randomly from L. Substitution changes `currentNode` to a randomly drawn function of F with the same arity. Deletion replaces `currentNode` with its largest child subtree, which often admits large deletion sub-operations.

The operator we consider here, HVL-Prime, functions slightly differently, since we restrict it to operate on trees where all functions take two parameters. Rather than choosing a node followed by an operation, we first choose one of the three sub-operations to perform. The operations then proceed as shown in Figure 1-1. Insertion and substitution are

exactly as in HVL; however, deletion only deletes a leaf and its parent to avoid the potentially macroscopic deletion change of HVL that is not in the spirit of bit-flip mutation. This change makes the algorithm more amenable to complexity analysis and specifies an operator that is only as general as our simplified problems require, contrasting with the generality of HVL, where all sub-operations handle primitives of any arity. Nevertheless, both operators respect the nature of GP's search among variable-length candidate solutions because each generates another candidate of potentially different size, structure, and composition.

In our analysis on these particular problems, we make one further simplification of HVL-Prime: substitution only takes place at the leaves. This is because our two problems only have one generic "join" function specified, so performing a substitution anywhere above the leaves is a vacuous mutation. Such operations only constitute one-sixth of all operations, so this change has no impact on any of the runtime bounds we derive.

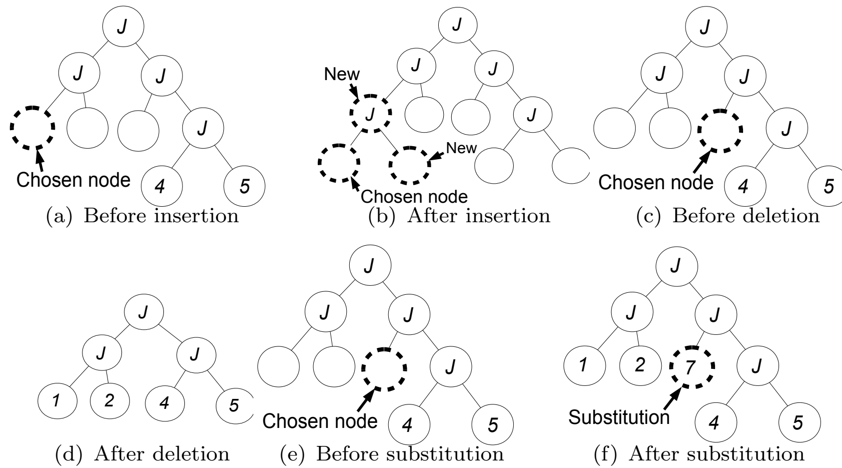

Figure 1-1. Example of the operators from HVL-Prime.

Algorithms

We define two genetic programming variants called (1+1) GP and (1+1) GP* . Both algorithms work with a population of size one and produce in each iteration one single offspring. (1+1) GP is defined in Figure 1-2 and accepts an offspring if it is as least as fit as its parent.

(1+1) GP* differs from (1+1) GP by accepting only solution that are strict improvements (see Figure 1-3).

1 Choose an initial solution X.

2 Set $X' := X$.

3 Mutate X' by applying HVL-Prime k times. For each application, randomly choose to either substitute, insert, or delete.

- If substitute, replace a randomly chosen leaf of X' with a new leaf $u \in L$ selected uniformly at random.
- If insert, randomly choose a node v in X' and select $u \in L$ uniformly at random. Replace v with a join node whose children are u and v, with the order of the children chosen randomly.
- If delete, randomly choose a leaf node v of X', with parent p and sibling u. Replace p with u and delete p and v.

4 If $f(X') \geq f(X)$, set $X := X'$.

5 Go to 2.

Figure 1-2. (1+1) GP

4'. If $f(X') > f(X)$, set $X := X'$.

Figure 1-3. Acceptance for (1+1) GP*

For each of (1+1) GP and (1+1) GP* we consider two further variants which differ in using one application of HVL-Prime ("single") or in using more than one ("multi"). For (1+1) GP-single and (1+1) GP*-single, we set $k = 1$, so that we perform one mutation at a time according to the HVL-Prime framework. For (1+1) GP-multi and (1+1) GP*-multi, we choose $k = 1 + \text{Pois}(1)$, so that the number of mutations at a time varies randomly according to the Poisson distribution.

We will analyze these four algorithms in terms of the expected number of fitness evaluations to produce an optimal solution for the first time. This is called the expected optimization time of the algorithm.

3. ORDER and MAJORITY

We consider two separable problems called ORDER and MAJORITY that have an independent, additive fitness structure. They both have multiple solutions, which we feel is a key property of a model GP problem because it holds generally for all real GP problems. They also both use the same primitive set, where $\bar{x}_i$ is the complement of x_i:

- $F := \{J\}$, J has arity 2.
- $L := \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$

The ORDER Problem

ORDER represents problems where the primitive sets include conditional functions, which gives rise to conditional execution paths. GP classification problems, for example, often employ a numerical comparison function (e.g. greater than X, less than X, or equal to X). This sort of function has two arguments (subtrees), one branch which will be executed only when the comparison returns true, the other only when it returns false (Koza, 1992). Thus, a conditional function results in a branching or conditional execution path, so the GP algorithm must identify and appropriately position the conditional functions to achieve the correct conditional execution behavior for all inputs.

ORDER is an abstracted simplification of this challenge: the conditional execution paths of a program are determined by tree inspection rather than execution. Instead of evaluating a condition test and then executing the appropriate condition body explicitly, an ORDER program's conditional execution path is determined by simply inspecting whether a primitive or its complement occurs first in an in-order leaf parse. Correct programs for the ORDER problem must express each positive primitive x_i before its corresponding complement $\bar{x}_i$. This correctness requirement is intended to reflect a property commonly found in the GP solutions

1 Derive conditional execution path P of X:

Init: l an empty leaf list, P an empty conditional execution path

1.1 Parse X inorder and insert each leaf at the rear of l as it is visited.

1.2 Generate P by parsing l front to rear and adding ("expressing") a leaf to P only if it or its complement are not yet in P (i.e. have not yet been expressed).

2 $f(X) = |\{x_i \in P\}|$.

Figure 1-4. $f(X)$ for ORDER

to problems where conditional functions are used: there exist multiple solutions, each with a set of different conditional paths. For example, for a tree X, with (after the inorder parse)

$$l = (x_1, \bar{x}_4, x_2, \bar{x}_1, x_3, \bar{x}_6), P = (x_1, \bar{x}_4, x_2, x_3, \bar{x}_6) \text{ and } f(X) = 3.$$

ORDER has the characteristic that whenever a solutions is non-optimal many successful mutation operators are possible to achieve an improvement in fitness. More precisely, let k be the fitness of a solution then there are $\Omega((n-k)^2)$ different mutation operators that lead to an improvement. Taking into a account the maximal tree size $T_{\max}$ during the run of the algorithm, one can show that the probability of an improvement is lower bounded by $\Omega\left(\frac{(n-k)^2}{nT_{\max}}\right)$ in this situation. As the fitness is not decreasing during the run of the algorithm, we can use the method of fitness-based partitions (see Chapter 4.2 in (Neumann and Witt, 2010) for an explanation of this method) to bound the runtime of our GP algorithms. This leads to the following results on the expected optimization time.

THEOREM 1.1 *The expected optimization time of the single-operation and multi-operation cases of (1+1) GP and (1+1) GP* on ORDER is* $O(nT_{\max})$ *in the worst case, where* n *is the number of* x_i *and* $T_{\max}$ *denotes the maximal tree size at any stage during the evolution of the algorithm.*

The MAJORITY problem

MAJORITY reflects a general (and thus weak) property required of GP solutions: a solution must have correct functionality and no incorrect functionality. Like ORDER, MAJORITY is a simplification that uses

1 Derive the combined execution statements S of X:

Init: l an empty leaf list, S is an empty statement list.

1.1 Parse X inorder and insert each leaf at the rear of l as it is visited.

1.2 For $i \leq n$: if count$(x_i \in l) \geq$ count$(\bar{x}_i \in l)$ and count$(x_i \in l) \geq 1$, add x_i to S

2 $f(X) = |S|$.

Figure 1-5. $f(X)$ for MAJORITY

tree inspection rather than program execution. A correct program in MAJORITY must exhibit at least as many occurrences of a primitive as of its complement and it must exhibit all the positive primitives of its terminal (leaf) set. Both the independent sub-solution fitness structure and inspection property of MAJORITY are necessary to make our analysis tractable.

For example, for a tree X, with (after the inorder parse)

$$l = (x_1, \bar{x}_4, x_2, \bar{x}_1, \bar{x}_3, \bar{x}_6, x_1, x_4), S = (x_1, x_2, x_4) \text{ and } f(X) = 3.$$

To solve the MAJORITY problem, a GP algorithm has to achieve for each i at least as many x_i as $\bar{x}_i$ variables. Therefore, it is crucial to analyze how to reduce the deficit of each variable according to the following definition.

Definition 1.2 *For a given GP tree, let $c(x_i)$ be the number of x_i variables and $c(\bar{x}_i)$ be the number of negated x_i variables present in the tree. For a GP tree representing a solution to the MAJORITY problem, we define the* deficit *in the* ith *variable by*

$$D_i = c(\bar{x}_i) - c(x_i).$$

Definition 1.3 *In a GP tree for MAJORITY, we say that x_i is* expressed *when $D_i \leq 0$ and $c(x_i) > 0$.*

The analysis considers the evolution of the deficits D_i over the course of the algorithm as n parallel random walks. It is shown that each positive D_i reaches zero at least as quickly as a balanced random walk, which is the condition for the corresponding x_i to be expressed; this, then, gives us the expected number of operations that we are required to perform on a particular variable before it is expressed.

These arguments lead to the following result for (1+1) GP-single.

THEOREM 1.4 *Let $D = \max_i D_i$ for an instance of MAJORITY initialized with T terminals drawn from a set of size $2n$ (i.e. terminals $x_1, ..., x_n, \bar{x}_1, ..., \bar{x}_n$). Then the expected optimization time of (1+1) GP-single is*

$$O(n \log n + DT_{\max} n \log \log n)$$

in the worst case.

Further results for ORDER and MAJORITY can be found in (Durrett et al., 2011). This, in particular, includes an average-case analysis for MAJORITY which shows that simple GP algorithms find solutions for this problem very quickly when the initial tree is chosen uniformly at random among all trees having a linear number of leaves.

4. The SORTING Problem

The problems ORDER and MAJORITY are in a sense easy as they have isolated problem semantics, and thus allow one to treat subproblems independently. The next step then is to consider problems that have dependent problem semantics, and we will do this in the following based on the SORTING problem. Sorting is one of the most basic problems in computer science. It is also 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). In (Scharnow et al., 2004), sorting is treated as an optimization problem where the task is to minimize the unsortness of a given permutation of the input elements. To measure unsortness, different fitness functions have been introduced and studied with respect to the difficulty of being optimized by permutation-based evolutionary algorithms.

In general, given a set of n elements from a totally ordered set, sorting is the problem of ordering these elements. We will identify the given elements by $1, \ldots, n$.

The goal is to find a permutation π_{opt} of $1, \ldots, n$ such that

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

holds, where $<$ is the order on the totally ordered set. W. l. o. g. we assume $\pi_{opt} = id$, i. e. $\pi_{opt}(i) = i$ for all i.

The set of all permutations π of $1, \ldots, n$ forms a search space that has already been investigated in (Scharnow et al., 2004) for the analysis of permutation-based evolutionary algorithms. The authors of this paper, investigate sorting as an optimization problem whose goal is to maximize

1 Derive a possibly incompletely defined permutation P of X:

Init: l an empty leaf list, P an empty list representing a possibly incompletely defined permutation

1.1 Parse X in order and insert each leaf at the rear of l as it is visited.

1.2 Generate P by parsing l front to rear and adding ("expressing") a leaf to P only if it is not yet in P, i. e. it has not yet been expressed.

2 Compute $f(X)$ based on P and the chosen fitness function.

Figure 1-6. Derivation of $f(X)$ for SORTING

the sortedness of a given permutation. We consider the following two fitness functions measuring the sortedness of a given permutation π.

- $INV(\pi)$, measuring the number of pairs in correct order,[1] which is the number of pairs (i, j), $1 \leq i < j \leq n$, such that $\pi(i) < \pi(j)$,
- $HAM(\pi)$, measuring the number of elements at correct position, which is the number indices i such that $\pi(i) = i$.

Considering tree-based genetic programming, we have to deal with the fact that certain elements are not present in a current tree. We extend our notation of permutation to incompletely defined permutations. Therefore, we use π to denote a list of elements, where each element of the input set occurs at most once. This is a permutation of the elements that occur in the tree. Furthermore, we use $\pi(x) = p$ to get the position p that the element x has within π. In the case that $x \notin \pi$, $\pi(x) = \perp$ holds. We adjust the definition of π to later accommodate the use of trees as the underlying data structure. For example, $\pi = (1, 2, 4, 6, 3)$ leads to $\pi(1) = 1$, $\pi(2) = 2$, $\pi(3) = 5$, $\pi(4) = 3$, $\pi(6) = 4$, and $\pi(5) = \perp$.

The basic idea behind for showing a runtime bound on INV is the following. It is always possible to increase the fitness by inserting a specific leaf at its correct position in order to achieve a fitness increment of at least 1. Therefore, the probability for such an improvement for each of our algorithms is $\Omega\left(\frac{1}{nT_{\max}}\right)$. Again $T_{\max}$ denotes the maximal size

[1]Originally, INV measures the numbers of pairs in wrong order. Our interpretation has the advantage that we need no special treatment of incompletely defined permutations.

of a tree during the run of the algorithm. Using the method of fitness-based partitions, and based on the observation that $n \cdot (n-1)/2+1$ fitness values are possible, the optimization time is upper bounded by $\sum_{k=0}^{n(n-1)/2} O\left(nT_{\max}\right)$ which leads to the following result.

THEOREM 1.5 *The expected optimization time of the single- and multi-operation cases of (1+1) GP* with INV is* $O(n^3 T_{\max})$.

For the sortedness measure HAM, we present a worst case example to demonstrate that the single- and multi-operation cases of (1+1) GP* can get stuck during the optimization process. Assuming that we initialize the algorithms with the following initial solution called T_w

$$\underbrace{n, n, \ldots, n}_{n+1 \text{ of these}}, 2, 3, \ldots, n-1, 1, n$$

it is easy to see that it is hard to achieve an improvement. It is clear that with a single HVL-Prime application, only one of the leftmost n can be removed. For an improvement in the sortedness, all $n+1$ leftmost leaves have to be removed in order for the rightmost n to become expressed. Additionally, 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 1. This cannot be done by the (1+1) GP*-single, resulting in an infinite runtime. However, (1+1) GP*-multi can improve the fitness, but 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 $e^{\Omega(n)}$.

THEOREM 1.6 *Let* T_w *be the initial solution to SORTING. Then the expected optimization time of (1+1) GP*-single and (1+1) GP*-multi is infinite respectively* $e^{\Omega(n)}$ *for the sortedness measure* HAM.

The proofs and further computational complexity results for genetic programming on the SORTING problem can be found in (Wagner and Neumann, 2011).

5. Future Directions

To conclude this chapter, we want to point out topics for future research. The results mentioned in the previous sections are initial ones and there are different ways to extend these studies. In the following, we

point out which are the most interesting directions for future research from our point of view.

Problems of Different Fitness Structure

The functions ORDER and MAJORITY can be seen as variants of the OneMax problem for binary strings. The characteristic of this problem is that each possible variable has the same weight and contributes independently to fitness. It would be interesting to analyze simplified GP algorithm on a much broader class where each variable contributes a different weight. A special case is linear scaling and exponential scaling of independent weights which has been studied experimentally in (Goldberg and O'Reilly, 1998). The general model we are thinking of matches the class of linear functions in the case of binary representations. The analysis of evolutionary algorithms working with binary representations carried out in (Droste et al., 2002) was a major breakthrough for the binary case and we expect that such results and the therefore developed methods will significantly pushed forward the theoretical understand of GP algorithms and set basis analysis of more complex problems.

Complexity versus accuracy

GP algorithms face the problem that the tree gets too large during the learning and optimization process. In the case of learning, it can often be observed that there is a trade-off between the accuracy of the learned function and the size of the tree (Gustafson et al., 2004). On the other hand, GP algorithms allow to express patterns several times in the tree although this does not increase the performs. Parsimony GP algorithms prefer trees of a smaller size if they have equal good performance. It would be interesting to study the impact of dealing with the complexity of the tree size in different ways. First, it would be interesting to study the impact of the parsimony approach and point out the impact on the runtime behaviour. Accepting only trees of lower complexity gives a clear search direction on plateaus, i. e. regions in the search space where solutions have equal good fitness. The aim would be to understand how this influences the run of a GP algorithm. One could point out the benefits and drawbacks of this approach in a rigorous way by computational complexity analyses.

The parsimony approach favours solutions of low complexity. Often users of genetic programming algorithms are interested in the trade-off between complexity and accuracy and want to obtain a set of solutions that gives them possible options according to these two objectives. Therefore, it would be interesting to analyze the trade-off between complexity and

accuracy that genetic programming algorithms observe when optimizing these two objective in a multi-objective approach. Our goal is understand how multi-objective genetic programming algorithms construct this set of trade-off solutions. For evolutionary algorithms working with binary representations it has been shown that the multi-objective model gives an additional search direction which allows to compute optimal solutions or good approximations much quicker than in the single-objective model (Neumann and Wegener, 2006; Friedrich et al., 2010). Therefore, it is interesting to study situations where the multi-objective formulation (complexity vs accuracy) is beneficial for the success of GP algorithms.

Populations, Crossover, and Diversity

One crucial parameter in a GP algorithm is the choice of the population size. Theoretical studies on the impact of the population size for evolutionary algorithms working on binary strings have shown that the choice of the population size can have a drastic impact on the runtime behaviour (Witt, 2006; Storch, 2008). Working with a population size of 1 leads to a hill-climbing behaviour that has the disadvantage of getting stuck in local optima. Therefore, often a larger population size is chosen to cope with this issue. The goal is to keep a diverse set of solutions during the run of the algorithm which allows to explore different regions of the search space. Closely connected to the choice of the population size, is the application of diversity mechanisms that ensure that a population consists of a diverse set of solutions. Such diversity mechanisms play a crucial role for the success of these algorithms (Burke et al., 2004). Therefore, our studies will concentrate on the impact of the population size in conjunction with commonly used diversity mechanisms. Diversity can be maintained by different mechanisms. One possibility is to compare the structure of the trees that constitute the population. Another possibility is to introduce a distance measure that is based on the number of subtrees that two solutions in the population share. Furthermore, diversity can be obtained by maintaining trees of different fitness. This allows to keep solutions of lower fitness which usually have a different structure than the best solutions in the population. It would be interesting to figure out the differences of these approaches by computational complexity analyses and show where they lead to significant different runtime results of such algorithms. Having solutions of different structure in the population is also crucial for successful crossover operators. Therefore, another topic of research is to study the impact of crossover in conjunction with the population size and appropriate diversity mechanisms.

6. Conclusions

The computational complexity analysis of genetic programming can provide new rigorous insight into the working principles of simplified genetic programming algorithms. In this chapter, we have pointed out the results of some initial studies. These studies show how to obtain computational complexity bounds for GP algorithms on problems with different characteristics. We outlined some topics for future research which would help to gain further rigorous insights into the behavior of genetic programming. We are optimistic that such computational complexity results can be obtained in the near future and that they will provide valuable new insights into this type of algorithms.

References

Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, San Francisco, CA, USA.

Burke, Edmund K., Gustafson, Steven, and Kendall, Graham (2004). Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62.

Castellini, Alexandre, Landa, Jorge, and Kikani, Jitendra (2004). Practical methods for uncertainty assessment of flow predictions for reservoirs with significant history – a field case study. Paper A-33, presented at the 9th European Conference on the Mathematics of Oil Recovery, Cannes, France, August 30 - September 2.

Deutsch, Clayton V. (2002). *Geostatistical Reservoir Modeling.* Oxford University Press.

Doerr, Benjamin and Happ, Edda (2008). Directed trees: A powerful representation for sorting and ordering problems. In *2008 IEEE World Congress on Computational Intelligence*, pages 3606–3613. IEEE Computational Intelligence Society, IEEE Press.

Droste, Stefan, Jansen, Thomas, and Wegener, Ingo (2002). On the analysis of the (1+1) evolutionary algorithm. *Theor. Comput. Sci.*, 276:51–81.

Durrett, Greg, Neumann, Frank, and O'Reilly, Una-May (2011). Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics. In *FOGA '11: Proceedings of the 11th ACM SIGEVO workshop on Foundations of Genetic Algorithms.* ACM. (to appear).

Friedrich, Tobias, He, Jun, Hebbinghaus, Nils, Neumann, Frank, and Witt, Carsten (2010). Approximating covering problems by randomized search heuristics using multi-objective models. *Evolutionary Computation*, 18(4):617–633.

Goldberg, David E. and O'Reilly, Una-May (1998). Where does the good stuff go, and why? how contextual semantics influence program structure in simple genetic programming. In Banzhaf, Wolfgang, Poli, Riccardo, Schoenauer, Marc, and Fogarty, Terence C., editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 16–36, Paris. Springer-Verlag.

Gustafson, Steven, Ekart, Aniko, Burke, Edmund, and Kendall, Graham (2004). Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 5(3):271–290.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

Neumann, Frank (2007). Expected runtimes of a simple evolutionary algorithm for the multi-objective minimum spanning tree problem. *European Journal of Operational Research*, 181(3):1620–1629.

Neumann, Frank and Wegener, Ingo (2006). Minimum spanning trees made easier via multi-objective optimization. *Natural Computing*, 5(3):305–319.

Neumann, Frank and Wegener, Ingo (2007). Randomized local search, evolutionary algorithms, and the minimum spanning tree problem. *Theor. Comput. Sci.*, 378(1):32–40.

Neumann, Frank and Witt, Carsten (2010). *Bioinspired Computation in Combinatorial Optimization – Algorithms and Their Computational Complexity.* Springer.

O'Reilly, Una-May (1995). *An Analysis of Genetic Programming.* PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada.

O'Reilly, Una-May and Oppacher, Franz (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Davidor, Yuval, Schwefel, Hans-Paul, and Manner, Reinhard, editors, *Parallel Problem Solving from Nature – PPSN III*, number 866 in Lecture Notes in Computer Science, pages 397–406, Jerusalem. Springer-Verlag.

Poli, Riccardo, Langdon, William B., and McPhee, Nicholas Freitag (2008). *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Poli, Riccardo, Vanneschi, Leonardo, Langdon, William B., and McPhee, Nicholas Freitag (2010). Theoretical results in genetic programming: the next ten years? *Genetic Programming and Evolvable Machines*, 11(3-4):285–320.

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

Storch, Tobias (2008). On the choice of the parent population size. *Evolutionary Computation*, 16(4):557–578.

Wagner, Markus and Neumann, Frank (2011). Computational complexity results for genetic programming and the sorting problem. *CoRR*, abs/1103.5797.

Witt, Carsten (2006). Runtime analysis of the (mu + 1) EA on simple pseudo-boolean functions. *Evolutionary Computation*, 14(1):65–86.

Yu, Tina, Wilkinson, Dave, and Castellini, Alexandre (2006). Constructing reservoir flow simulator proxies using genetic programming for history matching and production forecast uncertainty analysis. submitted.