Slightly beyond Turing-Computability for studying Genetic Programming

Olivier Teytaud

Slightly beyond Turing's computability for studying genetic programming

Olivier Teytaud

TAO, INRIA Futurs, LRI, UMR 8623 (CNRS - Univ. Paris-Sud),
olivier.teytaud@inria.fr

Abstract. Inspired by genetic programming (GP), we study iterative algorithms for non-computable tasks and compare them to naive models. This framework justifies many practical standard tricks from GP and also provides complexity lower-bounds which justify the computational cost of GP thanks to the use of Kolmogorov's complexity in bounded time.

1 Introduction

Limits of Turing-computability are well-known [28, 20]; many things of interest are not computable. However, in practice, many people work on designing programs for solving non-computable tasks, such as finding the shortest program performing a given task, or the fastest program performing a given task: this is the area of genetic programming (GP) [9, 13, 15]. GP in particular provides many human-competitive results (`http://www.genetic-programming.com/humancompetitive.html`), and contains 5440 articles by more than 880 authors according to the GP-bibliography [5]. GP is the research of a program realizing a given target-task roughly as follows:

1. generate (at random) an initial population of algorithms ;
2. select the ones that, after simulation, are "empirically" (details in the sequel) the most relevant for the target-task (this is dependent of a distance between the results of the simulation and the expected results, which is called the *fitness*) ;
3. create new programs by randomly combining and randomly mutating the ones that remain in the population ;
4. go back to step 2.

Theoretically, the infinite-computation models [6] are a possible model for studying programs beyond the framework of Turing Machines (TM). However, these models are far from being natural: they are intuitively justified by e.g. time steps with duration divided by 2 at each tick of the clock. We here work on another model, close to real-world practice like GP: iterative programs. These programs iteratively propose solutions, and what is studied is the convergence of the iterates to a solution in $\equiv_f$ with good properties (speed, space-consumption, size), and not the fact that after a finite time the algorithm stops and proposes a solution. The model, termed iterative model, is presented in algorithm 1. We

point out that we can't directly compare the expressive power of our model and the expressive power of usual models of computation; we here work in the framework of symbolic regression. Classically, programs work with a program as input, and output something (possibly a program). Here, the input can be a program, but it can also be made of black-box examples provided by an oracle. Therefore, we study algorithms with: (i) inputs, provided by an oracle (precisely, the oracle provides examples $(x_i, f(x_i))$); (ii) possibly, an auxiliary input, which is a program computing f; (iii) outputs, which are programs supposed to converge to the function f underlying the oracle *and* satisfying some constraints (e.g. asymptotically optimal size or asymptotically optimal speed). This is symbolic regression. We can encode decision problems in this framework (e.g., deciding if $\forall n, f(n) = 1$ by using x_i independent and $\forall n \in \mathbb{N}, P(x_i = n) > 0$); but we can not encode the problems as above as decision problems or other classical families of complexity classes. However, the links with various Turing degrees might be studied more extensively than in this paper.

Algorithm 1 GP - Iterative-algorithms

Set $p = 1$.
while true **do**
  Read an input $I_p = (x_p, y_p)$ with $y_p = f(x_p)$ on an oracle-tape.
  Perform standard computations of a TM (allowing reading and writing on an internal tape).
  Output some O_p.
  $p \leftarrow p + 1$
end while

This model is the direct translation of genetic programming in a Turing-like framework. A different approach consists in using also f as input. Of course, in that case, the important point is that O_p is "better" than f (faster, more frugal in terms of space-consumption, or smaller). We will see that at least in the general framework of f Turing-computable, the use of f as an input (as in algo. 2), and not only as a black-box (as in 1), is not so useful (theorem 3). We will in particular consider the most standard case, namely symbolic regression, i.e. inputs $I_1, \ldots, I_p, \ldots$ that are independently identically distributed on $\mathbb{N} \times \mathbb{N}$ according to some distribution. We assume that there exists some f such that with probability 1, $\forall i, f(x_i) = y_i$. Inputs are examples of the relation f (see e.g. [29, 10] for this model). The goal is that O_p converges, in some sense (see theorem 2 for details), to f. We will in particular study cases in which such a convergence can occur whereas without iterations f can not be built even with a black-box computing f as oracle; we will therefore compare the framework above (alg. 1) and the framework below (alg. 3). We point out that in algo. 3 we allow the program to both (i) read f on the input tape (ii) read examples (x_i, y_i) with $y_i = f(x_i)$. (i) allows automatic optimization of code and (ii) allows the use of randomized x. In cases in which (i) is allowed, what is interesting is

Algorithm 2 Iterative-algorithms using more than a black-box.

Set $p = 1$.
Read an input f (this is not black-box!).
while true **do**
  Read an input $I_p = (x_p, y_p)$ with $y_p = f(x_p)$ on an oracle-tape[1].
  Perform standard computations of a TM (allowing reading and writing on an internal tape).
  Output some O_p (there are therefore infinitely many outputs, but each of them after a finite time).
  $p \leftarrow p + 1$
end while

the convergence to some $g \equiv f$ such that g is in some sense "good" (frugal in terms of space or time). In contrast, algorithm 1 only uses examples. We will see that in the general case of Turing-computable functions f, this is not a big weakness (th. 3).

Algorithm 3 Finite-time framework.

Possibly read f on the input tape (if yes, this is not black-box).
Set $p = 1$.
while Some computable criterion **do**
  Possibly read an input $I_p = (x_p, y_p)$ on the input tape.
  Perform standard computations of a TM (allowing reading and writing on an internal tape).
  $p \leftarrow p + 1$
end while
Output some O.

In this spirit, we compare baseline standard (already known) results derived from recursion theory applied to finite-time computations (algo. 3), and results on iterative algorithms derived from statistics and optimization in a spirit close to GP (algo. 1). Interestingly, we will have theoretically-required assumptions close to the practice of GP, in particular (i) penalization and parsimony [26, 30, 16, 17], related to the bloat phenomenon which is the unexpected increase of the size of automatically generated programs ([11, 1, 14, 22, 19, 27, 2, 12]), and (ii) necessity of simulations (or at least of computations as expensive as simulations, see th. 3 for precise formalization of this conclusion). We refer to standard programs as "finite-time algorithms" (alg. 3), in order to emphasize the difference with iterative algorithms. Finite-time algorithms take something as input (possibly with infinite length), and after a finite-time (depending upon the entry), give an output. They are possibly allowed to use an oracle which provide 2-uples (x_i, y_i) with the x_i's i.i.d and $y_i = f(x_i)$. This is usually what we call an "algorithm". The opposite concept is iterative algorithms, which take something as input, and during an infinite time provide outputs, that are e.g. converging to the solution of

an equation. Of course, the set of functions that are computable in finite time is included in (and different from) the set of functions that are the limit of iterative algorithms (see also [21]). The (time or space) complexity of iterative algorithms is the (time or space) complexity of *one computation* of the infinite loop with one entry and one output. Therefore, there are two questions quantifying the overall complexity: the convergence rate of the outputs to a nice solution, and the computation time for each run through the loop. We study the following questions about GP:

- What is the natural formalism for studying GP ? We propose the algorithm 1 as a Turing-adapted-framework for GP-analysis.
- Can GP paradigms (algo. 1) outperform baseline frameworks (algo. 3) ? We show in theorem 2, contrasted with standard non-computability results summarized in section 3, that essentially the answer is positive.
- Can we remove the very expensive simulations from GP ? Theorem 3 essentially shows that simulation-times can not be removed.

2 Framework and notations

We consider TM [28, 20] with:

- one (read-only) binary input tape, where the head moves right if and only if the bit under the reading head has been read;
- one internal binary tape (read and write, without any restriction on the allowed moves);
- one (write-only) output binary tape, which moves of one and only one step to the right at each written bit.

The restrictions on the moves of the heads on the input and on the output tapes do not modify the expressive power of the TMs as they can simply copy the input tape on the internal tape, work on the internal tape and copy the result on the output tape. TM are also termed programs. If x is a program and e an entry on the input tape, then $x(e)$ is the output of the application of x to the entry e. $x(e) = \perp$ is the notation for the fact that x does not halt on entry e. We also let $\perp$ be a program such that $\forall e; \perp (e) = \perp$. A program p is a total computable function if $\forall e \in \mathbb{N}; p(e) \neq \perp$ (p halts on any input). We say that two programs x and y are equivalent if and only if $\forall e \in \mathbb{N}; x(e) = y(e)$. We denote this by $x \equiv y$. We let $\equiv_y = \{x; x \equiv y\}$.

All tapes' alphabets are binary. These TM can work on rational numbers, encoded as 2-uples of integers. Thanks to the existence of Universal TM, we identify TM and natural numbers in a computable way (one can simulate the behavior of the TM of a given number on a given entry in a computable manner). We let $< x_1, \ldots, x_n >$ be a n-uple of integers encoded as a unique number thanks to a given recursive encoding.

We use capital letters for programming-programs, i.e. programs that are aimed at outputting programs. There is no formal definition of a programming-program; the output can be considered as an integer; we only use this difference

for clarity. A decider is a total computable function with values in $\{0,1\}$. We denote by D the set of all deciders. We say that a function f recognizes a set F among deciders if and only if $\forall e; (e \in F \cap D \rightarrow f(e) = 1$ and $e \in D \setminus F \rightarrow f(e) = 0)$ (whatever may be the behavior, possibly f does not halt on e i.e. $f(e) = \perp$, for $e \notin D$). We let $\mathbf{1} = \{p; \forall e, p(e) = 1\}$, the set of programs always returning 1. The definition of the size $|x|$ of a program x is any usual definition such that there are at most 2^l programs of size $\leq l$. The space complexity is with respect to the internal tape (number of visited elements of the tape) *plus* the size of the program. We let (with a small abuse of notation as it depends on f and x and not only on $f(x)$) $time(f(x))$ (resp. $space(f(x))$) be the computation time (resp. the space complexity) of program f on entry x. $\mathbb{E}$ is the expectation operator. $Proba(.)$ is the probability operator ; by abuse, depending on the context, it is sometimes with respect to (x, y) and sometimes with respect to a sample $(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_m)$. Iid is a short notation for "independent identically distributed".

Section 3 presents non-computability results for finite-time algorithms. Section 4 which shows positive results for GP-like iterative algorithms. Section 5 studies the complexity of iterative algorithms. Section 6 concludes.

3 Standard case: finite time algorithms

We consider the existence of programs $P(.)$ such that when the user provides x, which is a Turing-computable function, the program computes $P(x) = y$, where $y \equiv x$ and y is not too far from being optimal (for size, space or time). It is known that for reasonable formalizations of this problem, such programs do not exist. This result is a straightforward extension of classical non-computability examples (the classical case is $C(a) = a$, we slightly extend this case for the sake of comparison with usual terminology in learning or genetic programming and in order to make the paper self-contained).

Theorem 1 (Undecidability). *Whatever may be the function $C(.)$ in $\mathbb{N}^{\mathbb{N}}$, there does not exist P such that for any total function x, $P(x)$ is equivalent to x and $P(x)$ has size $|P(x)| \leq C(\inf_{y \equiv x} |y|)$.*

Moreover, for any $C(.)$, for any such non-computable $P(.)$, there exists a TM using $P(.)$ as oracle, that solves a problem in $0'$, the jump of the set of computable functions.

Due to length constraints, we do not provide the proof of this standard result; but the proof is sketched in remark 1. We also point out without proof that using a random generator does not change the result:

Corollary 1 (No size optimization). *Whatever may be the function $C(.)$, there does not exist any program P, even possibly using a random oracle providing independent random values uniformly distributed in $\{0,1\}$ such that for any total function x, with probability at least $2/3$, $P(x)$ is equivalent to x and $P(x)$ has size $|P(x)| \leq C(\inf_{y \equiv x} |y|)$.*

The extension from size of programs to time complexity of programs requires a more tricky formulation than a simple total order relation "is faster than" ; a program can be faster than another for some entries and slower for some others. A natural requirement is that a program that suitably works provides a (at least nearly) Pareto-optimal program [18], i.e. a program f such that there's no program that is as fast as f for all entries, and better than f for some specific entry, at least within a tolerance function $C(.)$. The precise formulation that we propose is somewhat tricky but indeed very general; once again, we do not include the proof of this result (the proof is straightforward from standard non-computability result):

Corollary 2 (Time complexity). *Whatever may be the function $C(.)$, there does not exist any program P, even possibly using a random oracle providing independent random values uniformly distributed in $\{0,1\}$, such that for any total function x, with probability at least $2/3$,*

$P(x) \equiv x$ and there's no $y \equiv x$ such that y Pareto-dominates $P(x)$ (in time complexity) within $C(.)$, i.e. $\nexists y \in \equiv_x$ such that

$$\forall z; time(P(x)(z)) \geq C(time(y(z)))$$
$$and\ \exists z; time(P(x)(z)) > C(time(y(z)))$$

The result is also true when restricted to x such that a Pareto-optimal function exist.

After size (corollary 1) and time (corollary 2), we now consider space complexity (corollary 3). The proof in the case of space complexity relies on the fact that we include the length of programs in the space complexity.

Corollary 3 (Space complexity). *Whatever may be the function $C(.)$, there does not exist any program P, even possibly using a random oracle providing independent random values uniformly distributed in $\{0,1\}$, such that for any total function x, with probability at least $2/3$,*

$P(x) \equiv x$ and there's no $y \equiv x$ such that y dominates $P(x)$ (in space complexity) within $C(.)$, i.e., $\nexists y, y \equiv x$ and

$$\forall z; space(P(x)(z)) \geq C(space(y(z)))$$

$$and\ \exists z; space(P(x)(z)) > C(space(y(z)))$$

Remark 1 (Other fitnesses and sketch of the proofs). We have stated the non-computability result for speed, size and space. Other fitnesses (in particular, mixing these three fitnesses) lead to the same result. The key of the proofs above (th. 1, corollaries 1, 2, 3) is the recursive nature of sets of functions optimal for the given fitness, for at least the **1**-class of programs, which is a very stable feature. In results above, the existence of P, associated to this recursiveness in the case of **1**, shows the recursive nature of **1**, what is a contradiction.

4 Iterative algorithms

We recalled in section 3 that finite-time algorithms have deep limits. We now show that to some extent, such limitations can be overcome by iterative algorithms. The following theorem deals with learning deterministic computable relations from examples.

Theorem 2. *Assume that $y = f(x)$ where f is computable and $Proba(f(x) = \perp) = 0$ (with probability 1, f halts on x) and $\mathbb{E}time(f(x)) < \infty$. Assume that $(x_1, y_1), \ldots, (x_m, y_m)$ is an iid (independently identically distributed) sample with the same law as (x, y). We denote $ACT_m(g) = \frac{1}{m}\sum_{i=1}^{m} Time(g(x_i))$ and $c(a, b)$ any computable function, increasing as a function of $a \in \mathbb{Q}$ and increasing as a function of $b \in \mathbb{N}$, such that $\lim_{a\to\infty} c(a, 0) = \lim_{b\to\infty} c(0, b) = \infty$. We let*

$$f_m = P(< x_1, \ldots, x_m, y_1, \ldots, y_m >) \tag{1}$$

$$\text{and suppose that almost surely in the } x_i\text{'s,} \qquad \forall i; f_m(x_i) = y_i \tag{2}$$

and suppose that f_m is minimal among functions satisfying eq. 2 for criterion

$$c(ACT_m(f_m), |f_m|). \tag{3}$$

Then, almost surely in the x_i's, for m sufficiently large

$$Proba(P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)(x) \neq y) = 0. \tag{4}$$

Moreover, $c(\mathbb{E}time(f_m(x)), |f_m|)$ converges to the optimal limit:

$$c(\mathbb{E}time(f_m(x)), |f_m|) \to \inf_{f; Proba(f(x)\neq y)=0} c(\mathbb{E}time(f_m(x)), |f|) \tag{5}$$

and there exists a computable P computing f_m optimal for criterion 3 and satisfying eq. 2.

Proof:

The computability of f_m is established by the following algorithm:

1. Build a naive function h such that h terminates on all entries and $\forall i \in [[1, m]], h(x_i) = y_i$ (simply the function h that on entry x checks $x = x_i$ and replies y_i if such an x is found).

2. Consider a such that $c(a/m, 0) \geq c(ACT_m(h), |h|)$ and b such that $c(0, b) \geq c(ACT_m(h), |h|)$.

3. Define G as the set of all functions g with size $\leq b$.

4. Set $G' = G \setminus G''$, where G'' contains all functions in G such that $g(x_i)$ is not computed in time a or $g(x_i) \neq y_i$.

5. Select the best function in G' for criterion $c(ACT_m(g), |g|)$.

Any satisfactory f_m is in G and not in G'' and therefore is in G'; therefore this algorithms finds g in step 5.

We now show the convergence in eq. 5 and equation 4:

1. Let f^* be an unknown computable function such that $Proba(f^*(x) \neq y) = 0$, with $\mathbb{E}time f^*(x)$ minimal.

2. The average computation time of f^* on the x_i converges almost surely (by the strong law of large numbers). Its limit is dependent of the problem ; it is the expected computation time $\mathbb{E}f^*(x)$ of f^* on x.

3. By definition of f_m and by step 2, $f_m = P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)$ is such that $c(ACT_m(f_m), |f_m|)$ is upper bounded by $c(ACT_m(f^*), |f^*|)$, which is itself almost surely bounded above as it converges almost surely (Kolmogorov's strong law of large numbers [7]).

4. Therefore, f_m, for m sufficiently large, lives in a finite space of computable functions $\{f; c(0, |f|) \leq c(\sup_i ACT_i(f^*), |f^*|)\}$.

5. Consider $g_1, \ldots, g_k$ this finite family of computable functions.

6. Almost surely, for any $i \in [[1, k]]$ such that $Proba(g_i(x) \neq y) > 0$, there exists m_i such that $g_i(x_{m_i}) \neq y_{m_i}$. These events occur simultaneously as a finite intersection of almost sure events is almost sure ; so, almost surely, these m_i all exist.

7. Thanks to step 6, almost surely, for $m > \sup_i m_i$, $Proba(f_m(x) \neq y) = 0$.

8. Combining 5 and 7, we see that $f_m \in \arg\min_G c(ACT_m(g), |g|)$ where $G = \{g_i; i \in [[1, k]] \text{ and } Proba(g_i(x) \neq y) = 0\}$.

9. $c(ACT_m(g_i), |g_i|) \to c(\mathbb{E}time(g_i(x)), |g_i|)$ almost surely for any $i \in [[1, k]] \cap \{i; \mathbb{E}time(g_i(x)) < \infty\}$ as $c(.,.)$ is continuous with respect to the first variable (Kolmogorov's strong law of large numbers). As this set of indexes i is finite, this convergence is uniform in i.

10. $c(ACT_m(g_i), |g_i|) \to \infty$ uniformly in i such that $\mathbb{E}time(g_i(x)) = \infty$ as this set is finite.

11. Thanks to steps 9 and 10, $c(\mathbb{E}time(f_m(x)), |f_m|) \to \inf_{g; Proba(g(x) \neq y) = 0} c(\mathbb{E}time(g(x)), |g|)$. □

5 Complexity

We recalled above that finite time algorithms could not perform some given tasks (theorem 1, corollaries 1,2,3, remark 1). We have also shown that iterative methods combining size and speed are Turing-computable (theorem 2) and converge to optimal solutions. The complexity of Turing-computable programs defined therein (in theorem 2) is mainly the cost of simulation. We now show that it is not possible to avoid the complexity of simulation. This emphasizes the necessity of simulation (or at least, of computations with the same time-cost as simulation) for automatic programming in Turing spaces. Kolmogorov's complexity was introduced by Solomonov ([24]) in the field of artificial intelligence. Some versions include bounds on resource's ([24, 25, 8]), in particular on the computation-time ([3, 4, 23]):

Definition 1 (Kolmogorov's complexity in bounded time). *An integer x is T,S-complex if there is no TM M such that $M(0) = x \wedge |M| \leq S \wedge time(M(0)) \leq T$.*

Consider an algorithm A deciding whether an integer x is T, S-complex or not. Define $C(T,S)$ the worst-case complexity of this algorithm ($C(T,S) = \sup_x time(A(< x,T,S >))$). $C(T,S)$ implicitly depends on A, but we drop the dependency as we consider a fixed "good" A. Let's see "good" in which sense; there is some A which is "good" in the sense that with this A, $\forall T,S,\ C(T,S) < \infty$. This is possible as for x sufficiently large, x is T,S-complex, whatever may be its value; A does not have to read x entirely.

These notions are computable, but we will see that their complexity is large, at least larger than the simulation-parts. The complexity of the optimization of the fitness in theorem 4 is larger than the complexity $C(.,.)$ of deciding if x is T,S-complex ; therefore, we will lower bound $C(.,.)$.

Lemma 1 (The complexity of complexness). *Consider now T_n and some $S_n = O(\log(n))$, computable increasing sequences of integers computable in time $Q(n)$ where Q is polynomial. Then there exists a polynomial $G(.)$ such that*

$$C(T_n, S_n) > (T_n - Q(n))/G(n),$$

and in particular if T_n is $\Omega(2^n)$, $C(T_n,S_n) > \frac{T_n}{P(n)}$ where $P(.)$ is a polynomial.

Essentially, this lemma shows that, within polynomial factors, we can not get rid of the computation time T_n when computing T_n,S_n-complexity. The proof follows the lines of the proof of the non-computability of Kolmogorov's complexity by the so-called "Berry's paradox", but with complexity arguments instead of computability arguments. In short, we will use y_n, the smallest number that is "hard to compute".

Proof: Let y_n be the smallest integer that is T_n,S_n-complex.

Step 1: y_n is T_n,S_n-complex, by definition.

Step 2: But it is not $Q(n) + y_n \times C(T_n,S_n), C + D\log_2(n)$-complex, where C and D are constants, as it can be computed by (i) computing T_n and S_n (in time $Q(n)$) (ii) iteratively testing if k is T_n,S_n-complex, where $k = 1,2,3,\ldots,y_n$ (in time $y_n \times C(T_n,S_n)$.

Step 3: $y_n \leq 2^{S_n}$, as: (i) there are at most 2^{S_n} programs of size $\leq S_n$, (ii) therefore there are at most 2^{S_n} numbers that are not T_n,S_n-complex. (iii) therefore, at least one number in $[[0,2^{S_n}]]$ is T_n,S_n-complex.

Step 4: if $S_n = C + D\log_2(n)$, then y_n is upper bounded by a polynomial $G(n)$ (thanks to step 3).

Step 5: combining steps 1 and 2, $y_n C(T_n,S_n) > T_n - Q(n)$.

Step 6: using step 4 and 5, $C(T_n,S_n) > (T_n - Q(n))/G(n)$, hence the expected result. □

Consider now the problem $\mathcal{P}_{n,x}$ of solving in f the following inequalities:

$$Time(f(0)) \leq T_n,\ |t| \leq S_n,\ f(0) = x$$

Theorem 3 below shows that we can not get rid of the computation-time T_n, within a polynomial. This shows that using f in e.g. algo. 2 does not save up the simulation time that is requested in algorithm 4.

Theorem 3. *If $T_n = \Omega(2^n)$ and some $S_n = O(\log(n))$ are computable in polynomial time from n, then there exists polynomials $P(.)$ and $F(.)$ such that*

- *for any n and x, algorithm 4 solves problem $\mathcal{P}_{n,x}$ with computation-time at most $T_n F(n)$;*
- *there's no algorithm solving $\mathcal{P}_{n,x}$ for any n and x with computation-time at most $T_n/P(n)$.*

Proof: The computation time of algorithm 4 is straightforward. If an algorithm solves $\mathcal{P}_{n,x}$ for any x with computation-time at most $T_n/P(n)$, then this algorithm decides if x is T_n,S_n-complex in time at most $T_n/P(n)$, which contradicts lemma 1 if $P(n)$ is bigger than the polynomial of lemma 3. □

Algorithm 4 Algorithm for finding a program of size $\leq S$ and computation time $\leq T$ generating x.

```
for f of size ≤ S do
  Simulate T steps of f on entry 0.
  if output = x then
    Return f
    Break
  end if
end for
Return "no solution".
```

6 Conclusion

The iterative-model (algo. 1) is relevant for modeling GP in the sense that (i) it is very natural, as genetic programming tools work iteratively (ii) it reflects parsimony pressure (iii) by the use of Kolomogorov's complexity with bounded time, one can show that simulation as in genetic programming is necessary (at least the computation-time of simulation is necessary). (ii) and (iii) are typically formal elements in favor of genetic programming. Let's now sum up and compare our results, to see the relevance with the state of the art in genetic programming:

- In corollaries 1, 2, 3 we have shown that finite-time programming-programs can not perform the required task, i.e. finding the most efficient function in a space of Turing-equivalent functions. This, contrasted with th. 2, shows that algorithms as algo. 1 definitely can compute things that can not be computed by algorithms as algo 3.
- In theorem 2, we have shown that an iterative programming-program could asymptotically perform the required target-tasks, namely satisfying simultaneously (i) consistency, i.e. $f_m(x) = y$ with probability 1 on x and y as shown in eq. 4; (ii) good compromise between size and speed, as shown in

eq. 5. Interestingly, we need parsimony pressure in theorem 2 (short programs are preferred); parsimony pressure is usual in GP. This is a bridge between mathematics and practice. This leads to the conclusion that algo. 1 has definitely a larger computational power than algo. 3.
- The main drawback of GP is that GP is slow, due to huge computational costs, as a consequence of intensive simulations during GP-runs; but anyway one can not get rid of the computation time. In theorem 3, using a modified form of Kolmogorov's complexity, we have shown that getting rid of the simulation time is anyway not possible. This shows that the fact that f is not directly used, but only black-box-calls to f, in algo. 1, is not a strong weakness (at least within some polynomial on the computation time).

This gives a twofold theoretical foundation to GP, showing that (i) simulation + selection as in th. 2 outperforms any algorithm of the form of algo. 3 (ii) getting rid of the simulation time is not possible, and therefore using algo. 2 instead of 1 will not "very strongly" (more than polynomially) reduce the computational cost. Of course, this in the case of mining spaces of Turing-computable functions; in more restricted cases, with more decidability properties, the picture is very different. Refining comparisons between algorithms 1, 2, 3, is for the moment essentially an empirical research in the case of Turing-computable functions, termed genetic programming. The rare mathematical papers about genetic programming focus on restricted non-Turing-computable cases, whereas the most impressive results concern Turing-computable functions (also in the quantum case). This study is a step in the direction of iterative-Turing-computable models as a model of GP.

References

1. Wolfgang Banzhaf and William B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, 2002.
2. Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms Workshop at KI-94*, pages 33–38. Max-Planck-Institut für Informatik, 1994.
3. Harry Buhrman, Lance Fortnow, and Sophie Laplante. Resource-bounded kolmogorov complexity revisited. *SIAM Journal on Computing*, 2001.
4. L. Fortnow and M. Kummer. Resource-bounded instance complexity. *Theoretical Computer Science A*, 161:123–140, 1996.
5. S.M. Gustafson, W. Langdon, and J. Koza. Bibliography on genetic programming. In *The Collection of Computer Science Bibliographies*, 2007.
6. J. D. Hamkins. Infinite time turing machines. *Minds Mach.*, 12(4):521–539, 2002.
7. A. Y. Khintchine. Sur la loi forte des grands nombres. *Comptes Rendus de l'Academie des Sciences*, 186, 1928.
8. A.N. Kolmogorov. Logical basis for information theory and probability theory. *IEEE trans. Inform. Theory, IT-14, 662-664*, 1968.
9. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
10. G. Lugosi L. Devroye, L. Györfi. A probabilistic theory of pattern recognition, springer. 1997.

11. W. B. Langdon. The evolution of size in variable length representations. In *ICEC'98*, pages 633–638. IEEE Press, 1998.
12. W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In John Koza, editor, *Late Breaking Papers at GP'97*, pages 132–140. Stanford Bookstore, 1997.
13. W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.
14. W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. Angeline, editors, *Advances in Genetic Programming III*, pages 163–190. MIT Press, 1999.
15. William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.
16. Sean Luke and Liviu Panait. Lexicographic parsimony pressure. In W. B. Langdon et al., editor, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836. Morgan Kaufmann Publishers, 2002.
17. Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.
18. V. Pareto. *Manuale d'Economia Politica*. Milano: Societ Editrice, Libraria, 1906.
19. A. Ratle and M. Sebag. Avoiding the bloat with probabilistic grammar-guided genetic programming. In P. Collet et al., editor, *Artificial Evolution VI*. Springer Verlag, 2001.
20. H. Rogers. *Theory of recursive functions and effective computability*. McGraw-Hill, New York, 1967.
21. J. Schmidthuber. Hierarchies of generalized kolmogorov complexities and nonenumerable universal measures computable in the limit. *International Journal of Foundations of Computer Science 13(4):587-612*, 2002.
22. Sara Silva and Jonas Almeida. Dynamic maximum tree depth : A simple technique for avoiding bloat in tree-based gp. In E. Cantú-Paz et al., editor, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1776–1787. Springer-Verlag, 2003.
23. M. Sipser. A complexity theoretic approach to randomness. In *Proceedings of the 15th ACM Symposium on the Theory of Computing*, pages 330–335, 1983.
24. Ray Solomonoff. A formal theory of inductive inference, part 1. *Inform. and Control, vol. 7, number 1, pp. 1-22*, 1964.
25. Ray Solomonoff. A formal theory of inductive inference, part 2. *Inform. and Control, vol. 7, number 2, pp. 222-254*, 1964.
26. T. Soule and J. A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, 1998.
27. Terence Soule. Exons and code growth in genetic programming. In James A. Foster et al., editor, *EuroGP 2002*, volume 2278 of *LNCS*, pages 142–151. Springer-Verlag, 2002.
28. A. Turing. On computable numbers, with an application to the entscheidungsproblem. *proceedings of the London Mathematical Society, Ser. 2, 45, pp. 161-228 (reprinted in M. Davis (1965), The Undecidable, Ewlett, NY: Raven Press, pp. 155-222)*, 1936-1937.
29. V. Vapnik. The nature of statistical learning, springer. 1995.
30. B.-T. Zhang and H. Muhlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation, vol. 3, no. 1, pp. 17-38*, 1995.