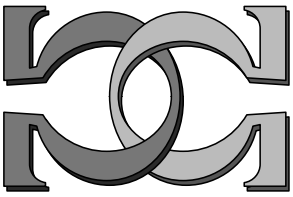

CDMTCS Research Report Series

Inductive Complexity Measures for Mathematical Problems

Mark Burgin[1],
Cristian S. Calude[2],
Elena Calude[3]

[1]University of California, Los Angeles, USA
[2]University of Auckland, NZ
[3]Massey University at Auckland,NZ

CDMTCS-416
30 December 2011

Centre for Discrete Mathematics and
Theoretical Computer Science

International Journal of Foundations of Computer Science

Inductive Complexity Measures for Mathematical Problems

Mark Burgin

Department of Mathematics, University of California, Los Angeles, USA
mburgin@math.ucla.edu

Cristian S. Calude

Department of Computer Science, University of Auckland, New Zealand
cristian@cs.auckland.ac.nz

Elena Calude

Institute of Natural and Mathematical Sciences, Massey University at Auckland, New Zealand
e.calude@massey.ac.nz

Received (Day Month Year)

Revised (Day Month Year)

Accepted (Day Month Year)

Communicated by (xxxxxxxxxx)

An algorithmic uniform method to measure the complexity of finitely refutable statements [9, 6, 7] was used to classify famous/interesting mathematical statements like Fermat's last theorem, the four colour theorem, and the Riemann hypothesis, [8, 15, 16]. Working with inductive Turing machines of various orders [1] instead of classical computations, we propose a class of inductive complexity measures and inductive complexity classes for mathematical statements which generalise the previous method. In particular, the new method is capable to classify Π_2–statements. As illustrations, we evaluate the inductive complexity of the Collatz and twin prime conjectures—statements which cannot be evaluated with the original method.

1. Introduction

Evaluating (or even guessing) the degree of complexity of an open problem, conjecture or proven mathematical statement is notoriously hard not only for beginners, but also for the most experienced mathematicians. The question is not trivial because mathematical problems can be so diverse: the Mathematics Subject Classification (MSC2000), based on two databases, Mathematical Reviews and Zentralblatt MATH, contains over 5,000 two-, three-, and five-digit classifications [25].

In a series of papers [9, 6, 7] a (uniform) algorithmic method to evaluate the complexity of mathematical problems was developed and, based on it, the complexity of various interesting mathematical sentences has been evaluated. The method, rooted in prefix complexity [5], can be applied to every finitely refutable statement when a single counter-example disproves the statement. This class includes all Π_1–statements, i.e. sentences of the form $\forall n\, P(n)$, where P is a computable predicate. Euclid's theorem of the infinity of primes, Goldbach's conjecture, Fermat's last theorem, Hilbert's tenth problem, the four colour theorem, the Riemann hypothesis, the integer partition theorem are examples of Π_1–statements. Clearly, not every mathematical statement is Π_1. The method cannot be applied to Collatz conjecture or to the twin prime

conjecture as currently it is not known how to express these conjectures as Π_1–statements (see the discussion in Section 3).

In this paper we introduce the *inductive complexity measures* and the *inductive complexity classes* for mathematical problems/statements which generalise the previous method. In particular, the new method is capable of classifying Π_2–statements, i.e. statements of the form $\forall n \exists i\, R(n,i)$, where $R(n,i)$ is a computable binary predicate. We illustrate our method by evaluating the inductive complexity of the Collatz and twin prime conjectures.

2. A semi-decidability complexity measure

An algorithm *semi-decides* a problem μ in case it returns 0 when μ is false and is undefined when μ is true.[a]

In [9, 6, 7] the complexity of a Π_1–problem π is defined by the size of the "smallest/simplest" program which systematically searches for a counter-example to π. A program which systematically searches for a counter-example to π semi-decides π because the program stops if and only if there exists an m such that $P(m)$ is false; in particular, if π is true, the program never stops. Accordingly, the resulting complexity is a semi-decidability complexity measure.

To obtain a formal definition we fix a model of computation, in our case a universal prefix-free Turing machine U. The requirement of prefix-freeness is motivated by coding theory (halting programs form an instantaneous code) and the relation to the binary expansion of Omega number [5] (to be discussed later in this section). We use register machine programs, cf. [9, 6, 7], to construct a machine U (which is fully described in [7]); for the present goal, U has to be *minimal* in the sense that none of its instructions can be simulated by a program for U written with the remaining instructions.

To every representation of the Π_1–problem as $\pi = \forall m P(m)$, where P is a computable predicate, we associate the *algorithm* $\Pi_P = \inf\{n : P(n) = \text{ false}\}$ which "tries" to find the smallest n such that $P(n)$ is false; if P is true, then Π_P runs forever, so no number is produced.

There are many programs (for U) which implement Π_P; without loss of generality, any such program will be denoted also by Π_P. The simplest way to write a program (in pseudo-code) for the algorithm Π_P is as follows:

```
set n to 1; while (P(n) = true) set n to n+1; print n; stop.
```

Obviously, this algorithm may not be the smallest in size.

The *semi-decidability complexity* (with respect to U) of a Π_1–problem π is defined by the length of the smallest-length program (for U) for the algorithm Π_P—defined as above—where minimisation is calculated for all possible representations $\pi = \forall n P(n)$ and all programs Π_P:

$$C_U(\pi) = \min\{|\Pi_P| : \pi = \forall n P(n)\}. \tag{1}$$

Note that π is true iff $U(\Pi_P)$ never halts.

The search for a counter-example uses essential "knowledge" about π, but not "the deep understanding an expert mathematician may have about π". This "knowledge" evolves in time as more "understanding" of the problem is accumulating, hence simpler predicates P for describing π are found. Some computable predicates P' such that $\pi = \forall m P'(m)$ are not "genuine". For example, if π is true, we can take P' to be the constant true predicate, which is not genuine in this context. Why? Because the representation $\pi = \forall m P'(m)$ does not reflect any "algorithmic

[a]The notions of semi-decidability and decidability (to be discussed in the next section) have been introduced and studied in [4] in the general context of the axiomatic theory of algorithms.

knowledge" about π permitting a "real" search for a counter-example, say by a proof-assistant, a search that should work only for π. Consequently, for every computable predicate P, the program containing the single instruction halt is not a program for Π_P. This requirement seems difficult to capture in mathematical formulas, but is easy to recognise. We have a few proposals to formalise "a search for a counter-example which is unique for π" none of which seems "general enough", so without loss of generality this condition will be kept informal.

In this way we can uniformly and objectively compare problems from different areas of mathematics, a task generally impossible for human experts. For C_U it is irrelevant whether π is known to be true or false. However, the comparison is limited by the uniformity of the solution we analyse—we evaluate only one possible solution from infinitely many candidates. Knowing the complexity of an open Π_1–problem does not give any clue about other ways to solve the problem.

Another way of defining the semi-decidability complexity C_U is to consider the Omega number associated to U $\Omega_U = \sum_{U(x) \text{ halts}} 2^{-|x|}$. It is known (see [5]) that Ω_U is Martin-Löf random (hence incomputable) and the halting problem with respect to U for all programs p with $|p| \leq N$ can be solved using the first N bits of the binary expansion of Ω_U. The complexity $C_U(\pi)$ is the smallest N such that given the first N bits of the binary expansion of Ω_U one can decide whether π is true or not.

Because the complexity C_U is incomputable, we work with upper bounds for C_U. As the exact value of C_U is not important, following [7] we classify Π_1–problems into the following classes:

$$\mathfrak{C}_{U,n} = \{\pi \,:\, \pi \text{ is a } \Pi_1\text{–problem}, C_U(\pi) \leq 2^{10} n\}.$$

The threshold $2^{10}n$ is to some extent arbitrary and may be easily changed; its main goal is only to provide a scale to compare/rank mathematical statements in a uniform way. Here is one argument in favour of our choice. If instead of U we use a different universal prefix-free Turing machine U' then one can compute a constant c (depending upon U and U') such that for every Π_1–problem π one has $|C_U(\pi) - C_{U'}(\pi)| \leq c$. Experimental calculation shows that for minimal machines the constant c is smaller than 2^{10}.

Here are some results obtained with this method. Legendre's conjecture (there is a prime number between n^2 and $(n+1)^2$, for every positive integer n), Fermat's last theorem (there are no positive integers x, y, z satisfying the equation $x^n + y^n = z^n$, for any integer value $n > 2$) and Goldbach's conjecture (every even integer greater than 2 can be expressed as the sum of two primes) are in $\mathfrak{C}_{U,1}$, Dyson's conjecture (the reverse[b] of a power of two is never a power of five) is in $\mathfrak{C}_{U,2}$ [6, 7, 16], the Riemann hypothesis (all non-trivial zeros of the Riemann zeta function have real part 1/2) and the integer partition theorem (the number of partitions of an integer into odd integers is equal to the number of partitions into distinct integers) are each in $\mathfrak{C}_{U,3}$ [15, 10], and the four colour theorem (the vertices of every planar graph can be coloured with at most four colours so that no two adjacent vertices receive the same colour) is in $\mathfrak{C}_{U,4}$ [8]. Related results have appeared in [12]. For every positive integer n there is an integer $m > n$ such that $\mathfrak{C}_{U,n}$ is strictly included in $\mathfrak{C}_{U,m}$; it is an open question whether m can be taken to be $n+1$. Except for problems in $\mathfrak{C}_{U,1}$, all other results *are not known to be strict.*

3. The Collatz conjecture

The Collatz conjecture[c] proposed by L. Collatz (when he was a student) is the following: given any positive integer seed a_1, there exists a natural N such that $a_N = 1$, where

$$a_{n+1} = \begin{cases} a_n/2, & \text{if } a_n \text{ is even,} \\ 3a_n + 1, & \text{otherwise.} \end{cases}$$

[b]The reverse of a number is the number formed with the same digits but written in opposite order. For example, the reverse of 131072 is 270131.

[c]Also known as the Syracuse conjecture, the $3x+1$ problem, Kakutani's problem, Hasse algorithm, or Ulam's problem.

There is a huge literature on this problem and various natural generalisations: see [22, 19, 17, 23]. Erdös is quoted in [22] by saying that *Mathematics may not be ready for such problems.* To be able to evaluate the semi-decidability complexity measure of the Collatz conjecture we need to effectively construct a program Π_{Collatz} such that Collatz's conjecture is false if and only if Π_{Collatz} halts; the mere existence of such a program won't be enough.

A brute-force tester, i.e. the program which enumerates all seeds and for each of them tries to find an iteration equal to 1, may never stop for two different reasons: a) because the Collatz conjecture is true, b) because there exists a seed a_1 such that there is no N such that $a_N = 1$. How can one algorithmically differentiate between these cases? How can one refute b) by a brute-force test? We don't know the answers to the above questions. However, a simple non-constructive argument [9] shows that the Collatz conjecture is a Π_1–statement. Indeed, observe that the set

$$Collatz = \{a_1 \mid \text{ there exists } N \geq 1 \text{ such that } a_N = 1\}$$

is computably enumerable. Collatz's conjecture requires to prove that the set $Collatz$ coincides with the set of all positive integers.

If $Collatz$ is not computable, then the conjecture is false, and any program which eventually halts can be taken as Π_{Collatz} as a) is ruled out.[d] If $Collatz$ is computable, then we can write a program Π_{Collatz} to find an integer not in $Collatz$: the conjecture is true if and only if Π_{Collatz} never stops.

In fact, the above reasoning works for every Π_2–statement, i.e. a statement of the form $\forall n \exists i R(n, i)$, for some computable binary predicated R (the Collatz conjecture is a Π_2–statement).

The above observation shows that although, in principle, the developed method can be applied to the Collatz conjecture and, in fact, to many, but not all Π_2–statements, it is impossible to do this (at least for the time being) as we do not know how to explicitly construct the program Π_{Collatz}. This raises the question of finding a more general method which can be applied to the Collatz conjecture and similar mathematical statements.

4. Inductive complexity measures of order k

Can one extend the complexity method developed for Π_1–statements to Π_2–statements? The algorithmic direct verification of the validity of a Π_2–statement does not work as it never stops irrespective whether the statement is true or false.

However, there is a general method for testing the Π_2–statement $\rho = \forall n \exists i R(n, i)$, where R is a decidable binary predicate. The program has two loops: the external loop iterates over n, the internal loop iterates over i, and just before starting the evaluation of $R(n, i)$, the program outputs n. If ρ is true the output value keeps changing and if ρ is false then the value gets stuck at the first n for which $R(n, i)$ is false for all i.

The above method cannot be programmed with a classical algorithm or Turing machine, but can be formally presented in terms of inductive computations [1] (which are, by Shöenfield's limit lemma [24], computations with oracle access to the Halting Set).

Consider a Turing machine M with input, working and output tapes. The result of a classical computation of M on input x is the content of the output tape in case the computation stops; if the computation continues forever, there is no result. The result of an *inductive computation* of M on input x is the content of the output tape in case this content stops changing at some step of the computation; otherwise, there is no result [1]. A Turing machine running inductive computations is called an *inductive Turing machine of first order.*

[d]Such programs won't be "genuine" in the sense discussed in Section 2.

The hardware is the same for classical and inductive computations; the difference is in the semantics. In contrast with the classical computation of M on x—which assumes that the computation has stopped for the result to be produced—the inductive computation of M on x may not stop but still produce a result in case the content of the output tape has stabilised at some step of the (possibly infinite) computation. Of course, halting computations produce the same result classically and inductively.

The functional composition $T_1(T_2)$ of two inductive Turing machines of first order is not necessarily an inductive Turing machine of first order, so it is called an *inductive Turing machine of second order*. An *inductive Turing machine of order* $k \geq 2$ is the functional composition of k inductive Turing machines of first order (formal details are presented in [3]). If M is a Turing machine, then by M^{ind} we denote the machine working inductively; inductive Turing machines of order $k \geq 1$ are denoted by $M^{ind,k}$. Inductive programs will be similarly denoted.

The method of evaluating the semi-decidability complexity can be reformulated in terms of inductive Turing machines of first order. To the computable predicate $P(m)$ we assign the problem $\pi = \forall m P(m)$ and the algorithm $\Pi_P = \inf\{n : P(n) = \text{ false}\}$. It is easy to construct an inductive program of first order $\Pi_P^{ind,1}$ such that $||\Pi_P| - |\Pi_P^{ind,1}||$ is bounded by a small constant[e]:

$$\pi \text{ is true if and only if } U(\Pi_P) \text{ never stops if and only if } U^{ind}(\Pi_P^{ind,1}) = 0.$$

The *inductive complexity measure of first order* is defined by

$$C_U^{ind,1}(\pi) = \min\{|\Pi_P^{ind,1}| : \pi = \forall n P(n)\}, \tag{2}$$

and the corresponding *inductive complexity class of first order* by

$$\mathfrak{C}_{U,n}^{ind,1} = \{\pi : \pi \text{ is a } \Pi_1\text{–statement}, C_U^{ind,1}(\pi) \leq 2^{10} n\}. \tag{3}$$

There is a constant $c < 2^{10}$ such that for every Π_1–statement π we have:

$$|C_U(\pi) - C_U^{ind,1}(\pi)| \leq c,$$

hence

$$\mathfrak{C}_{U,n} \subseteq \mathfrak{C}_{U,n}^{ind,1} \subseteq \mathfrak{C}_{U,n+1}. \tag{4}$$

In this way all results proved for C_U and $\mathfrak{C}_{U,n}$ translate automatically in results for $C_U^{ind,1}$ and $\mathfrak{C}_{U,n}^{ind,1}$.

Why do we need to compute inductively instead of classically? First, because we can extend the first method from Π_1–sentences to more complex sentences, in particular, to Π_2–sentences. Secondly, because in contrast with the semi-decidability complexity measure C_U, the inductive complexity measure of first order is a decidability complexity measure. (Recall that an algorithm *decides* a problem ρ in case it returns 0 when ρ is false and 1 when ρ is true; a decidability complexity measures the complexity of an algorithm deciding the problem.) This has the advantage that the complexity of a sentence ρ is the same as the complexity of the negation of ρ. Finally, the inductive computation goes beyond the Turing barrier, in the sense it can compute Turing incomputable functions, and the possibility of effectively running such a computation is not completely elucidated (at the time of the writing of this paper). Does this create a problem? The answer is

[e]For example, in the language described in Section 5, $\Pi_P^{ind,1}$ is constructed as follows: in the program Π_P the stop instruction % is replaced with the instruction `&a, 1` followed by %; here `a` is a register not appearing in Π_P, which was designed as output register.

negative as the main goal here is not to solve the problem, i.e. not to run the computation, but to encode efficiently an algorithm solving the problem.[f]

From the Π_2–sentence $\forall n \exists i R(n,i)$ we construct the inductive Turing machine of first order $T_R^{ind,1}$ defined by

$$T_R^{ind,1}(n) = \begin{cases} 1, \text{ if } \exists i R(n,i), \\ 0, \text{ otherwise}. \end{cases} \tag{5}$$

Next we construct the inductive Turing machine of second order[g] $M_R^{ind,2}$ defined by

$$M_R^{ind,2} = \begin{cases} 1, \text{ if } \forall n \exists i R(n,i), \\ 0, \text{ otherwise}. \end{cases} = \begin{cases} 1, \text{ if } \forall n\, (T_R^{ind,1}(n) = 1), \\ 0, \text{ otherwise}. \end{cases} \tag{6}$$

Note that the predicate $T_R^{ind,1}(n) = 1$ is well-defined because the inductive Turing machine of first order $T_R^{ind,1}$ always produces an output. However, the inductive Turing machine $M_R^{ind,2}$ is of the *second order* because it uses an inductive Turing machine of the first order $T_R^{ind,1}$.

To every mathematical sentence of the form $\rho = \forall n \exists i R(n,i)$, where $R(n,i)$ is a computable predicate, we associate the inductive Turing machine of second order $M_R^{ind,2}$ as above. Note that there are many programs for U^{ind} which implement $M_R^{ind,2}$; for each of them we have:

$$\forall n \exists i R(n,i) \text{ is true if and only if } U^{ind}(M_R^{ind,2}) = 1. \tag{7}$$

In this way, the inductive complexity measure of first order $C_U^{ind,1}(\pi)$ (see (3)) can be extended to the *inductive complexity measure of second order*:

$$C_U^{ind,2}(\rho) = \min\{|M_R^{ind,2}| \,:\, \rho = \forall n \exists i R(n,i)\},$$

and the inductive complexity class of first order $\mathfrak{C}_{U,n}^{ind,1}$ (see (3)) to the *inductive complexity class of second order*:

$$\mathfrak{C}_{U,n}^{ind,2} = \{\rho \,:\, \rho = \forall n \exists i R(n,i), C_U^{ind,2}(\rho) \leq 2^{10} n\}.$$

The above construction of the inductive Turing machine of second order $M_R^{ind,2}$ leading to the equivalence (7) is "algorithmic". The optimization necessary for the approximation of the complexity $C_U^{ind,2}(\rho)$, on the other hand, is not algorithmic: it depends on the predicate $R(n,i)$ and requires some creativity to discover.

Based on the above construction and results in [2] one gets the following facts:

(1) for every $n, t \geq 1$ there exists $m \geq n$ such that $\mathfrak{C}_{U,n}^{ind,t} \subset \mathfrak{C}_{U,m}^{ind,t}$,
(2) there exist Π_2–statements ρ which are not in any class $\mathfrak{C}_{U,n}^{ind,1}$, $n \geq 1$,
(3) the semi-decidability problem of every Π_2–statement ρ is in a class $\mathfrak{C}_{U,n}^{ind,1}$, for some $n \geq 1$.

Working with inductive Turing machines of order $k > 2$ we can define the *inductive complexity measure of order k* and the *inductive complexity class of order k*.

[f]Incidentally, this is an example of the use of a hypercomputation model for which the problem whether the computation can be materially executed in our Universe is irrelevant.
[g]This machine solves the decidability of the Collatz conjecture and has no input argument.

5. A universal prefix-free binary Turing machine

The register machine language we use is a refinement of the language in [9], constructed in [7]; see also [18]. The register machine language is simple and minimal (each instruction is essential, that is no instruction can be reduced to a combination of the other instructions). It consists of the following instructions:

= R1,R2,R3 If the content of R1 and R2 are equal, then the execution continues at the $R3^{rd}$ instruction of the program. If the contents of R1 and R2 are not equal, then execution continues with the next instruction in sequence.

& R1,R2 The content of register R1 is replaced by R2.

+ R1,R2 The content of register R1 is replaced by the sum of the contents of R1 and R2.

! R1 One bit is read into the register R1, so the content of R1 becomes either 0 or 1. Any attempt to read past the last data-bit results in a run-time error. [h]

% This is the last instruction for each register machine program before the input data. It halts the execution in two possible states: either successfully halts or it halts with an under-read error.

A register machine program is a finite list of these instructions. It is allowed access to an arbitrary number of registers, and each register can hold an arbitrarily large positive integer. The prefix free binary encoding of these instructions is discussed in detail in [6, 7], and it is briefly presented below.

Selecting one or several registers as output registers, register machine programs can compute not only in the classical mode (the program has to halt to get a result), but also in the inductive mode. In the inductive mode, register machine programs can simulate inductive Turing machines of the first order in the same way they can simulate Turing machines working in classical mode.

With subprograms that work in the inductive mode it is possible to obtain register machines that can simulate inductive Turing machines of higher orders.

6. Binary coding of programs

In this section we develop a systematic efficient method to uniquely code in binary the register machine programs; for consistency with previous results we use the same prefix-free coding method as in [6, 7, 10, 11].

Each instruction has its own binary op-code, registers names are encoded as the string $\text{code}_1 = 0^{|x|}1x,\ x \in \{0,1\}^*$ and literals are encoded $\text{code}_2 = 1^{|x|}0x,\ x \in \{0,1\}^*$. Some instructions can take registers or literals, but this encoding gives an unambiguous distinction between the two options. The encodings are summarised below:

(1) `& R1,R2` is coded in two different ways depending on R2: $01\text{code}_1(\text{R1})\text{code}_i(\text{R2})$, where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.
(2) `+ R1,R2` is coded in two different ways depending on R2: $111\text{code}_1(\text{R1})\text{code}_i(\text{R2})$, where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

[h]This instruction is not used in our codes, but it is necessary for the universality of the register machine language.

(3) `= R1,R2,R3` is coded in four different ways depending on the data types of R2 and R3: $00\text{code}_1(\text{R1})\text{code}_i(\text{R2})\text{code}_j(\text{R3})$, where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer, $j = 1$ if R3 is a register and $j = 2$ if R3 is an integer.
(4) `!R1` is coded by $110\text{code}_1(\text{R1})$.
(5) `%` is coded by 100.

All codings for instruction names and special symbol comma, registers and non-negative integers are prefix-free; the prefix-free codes used for registers and non-negative integers are disjoint. The code of any instruction is the concatenation of the codes of the instruction name and the codes (in order) of its components, hence the set of codes of instructions is prefix-free. The code of a program is the concatenation of the codes of its instructions, so the set of codes of all programs is prefix-free too.

7. Pseudo-code for the inductive Turing machine $M_R^{ind,2}$

In this section we present the pseudo-code for the inductive Turing machine of second order $M_R^{ind,2}$ defined in (6). The inductive Turing machine $M_R^{ind,2}$—denoted by `Main`—depends on the inductive Turing machine of first order $T_R^{ind,1}(n)$ denoted by `H(n)`, which in turn is a function of the binary computable predicate R: see (5) and (6). The result is output in the register `OM`. This notation will be used also in the following two sections in which concrete forms of `H` will be discussed.

```
   Main (H)
1. set OM to 1
2. set n to 1
3.    if H(n)=1
4.            then set n to n+1
5.                 goto 3
6.            else set OM to 0
7.                 goto 8
8. stop
```

8. Inductive complexity of the Collatz conjecture

In this section we evaluating the inductive complexity of second order of the Collatz conjecture.

We start with the function

$$T_R^{ind,1}(n) = \begin{cases} 1, \text{ if } \exists i(F^i(n) = 1), \\ 0, \text{ otherwise}, \end{cases}$$

where

$$F(x) = \begin{cases} x/2, & \text{if } x \text{ is even}, \\ 3x + 1, & \text{otherwise}, \end{cases}$$

and F^i is the ith iteration of F.

Based on it we define the inductive Turing machine$M_{\text{Collatz}}^{ind,2}$ by the formula (6). The pseudo-code for `H(n) = HCollatz(n)` uses `G` to compute $F^i(n)$:

```
   HCollatz(n)
1.  set OH to 0.
2.  set N to n
```

```
3.  set i to 1
4.  set k to 1
5.  if N is even
6.       then set G to N/2
7.            goto 9
8.       else set G to 3*N+1
9.  if k = i
10.      then if G = 1
11.             then set OH to 1
12.                  stop
13.             else set i to i+1
14.                  goto 4
15.      else set k to k+1
16.           set N to G
17.           goto 5
```

The inductive program solving the Collatz conjecture based on $M^{ind,2}_{\text{Collatz}}$ is presented in Table 1. The program uses the routine HColatz and has 38 instructions and 516 bits, therefore the Collatz conjecture is in the inductive complexity class $\mathfrak{C}^{ind,2}_{U,1}$.

label	instruction	label	instruction	label	instruction
	=a,a,MAIN		+g,a		=a,a,c
HCollatz	&OH,0		+g,a	MAIN	&OM,1
	&i,1		+g,1		&N,1
LH7	&k,1	LH8	=k,i,LH4	LM1	&a,N
LH5	&e,0		+k,1		&c,LM2
	&f,0		&a,g		=a,a,HColatz
LH1	=a,e,LH3		=a,a,LH5	LM2	=OH,1,LM3
	+e,1	LH4	=g,1,LH6		&OM,0
	=a,e,LH2		+i,1		=a,a,LM4
	+e,1		=a,a,LH7	LM3	+N,1
	+f,1	LH3	&g,f		=a,a,LM1
	=a,a,LH1		=a,a,LH8	LM4	%
LH2	&g,a	LH6	&OH,1		

Table 1. Inductive program for the Collatz conjecture

9. Inductive complexity of the twin prime conjecture

In this section we evaluate the inductive complexity of second order of the twin prime conjecture: there are infinitely many primes p such that $p+2$ is also prime. Using the procedure in Section 7 for the twin prime conjecture we present the pseudo-code for `H(n) = HTwinPrime(n)`:

```
   HTwinPrime(n)
1.  set OH to 0
2.  set i to n
3.  if i is prime
4.      then if i+2 is prime
5.             then set OH to 1
6.                  goto 10
7.             else set i to i+1
8.                  goto 3
```

```
9.      else goto 7
10. stop
```

The inductive program solving the twin prime conjecture is presented in Table 2.

label	instruction	label	instruction	label	instruction
	=a,a,MAIN		=a,a,LP7		&OH,1
PRIME	&f,2	LP6	&d,1		&a,ah
LP1	=f,a,LP6	LP7	=a,a,c		&c,ch
	&e,f	HTwinPrime	&ah,a		=a,a,c
	&h,0		&ch,c	MAIN	&OM,1
LP2	=e,a,LP3		&OH,0		&N,1
	+e,1		&ih,a	LM1	&a,N
	+h,1	LH1	&c,LH2		&c,LM2
	=h,f,LP4		&a,ih		=a,a,HTwinPrime
	=a,a,LP2		=a,a,PRIME	LM2	=OH,1,LM3
LP3	=h,0,LP5	LH2	=d,1,LH4		&OM,0
	+f,1	LH3	+ih,1		=a,a,LM4
	=a,a,LP1		=a,a,LH1	LM3	+N,1
LP4	&h,0	LH4	&c,LH5		=a,a,LM1
	=a,a,LP2		=a,a,PRIME	LM4	%
LP5	&d,0	LH5	=d,0,LH3		

Table 2. Inductive program for the twin prime conjecture

The predicate PRIME (PRIME$(a) = 1$ if a is prime and $= 0$ otherwise) is implemented in the 18 instructions starting with PRIME &f,2 and its result is stored in register d. The main program has 47 instructions and 649 bits, therefore is in the inductive complexity class $\mathfrak{C}_{U,1}^{ind,2}$.

10. Conclusions

In this paper we have extended the method of evaluation of the complexity of mathematical problems based on their semi-decidability to a more general and powerful method based on the decidability of problems. In this process we have replaced classical computations by inductive computations [3] to define the inductive complexity and the inductive complexity classes. We have illustrated our method by evaluating the inductive complexity of the decidability of the Collatz and twin prime conjectures. These problems cannot be currently evaluated with the semi-decidability complexity developed in [9, 6, 7, 16, 15]; they are both in the lowest inductive complexity class $\mathfrak{C}_{U,1}^{ind,2}$. In [11] and [20] it was proved that the P vs NP problem and Goodstein's theorem (every "Goodstein sequence" eventually terminates at 0) are both in the inductive complexity class of second order 7, $\mathfrak{C}_{U,7}^{ind,2}$.

One of the referees noticed that

> Goodstein's theorem is relatively easy to prove, whereas the Collatz conjecture is apparently much more difficult. This intuition is in contradiction with the estimated complexities: Goodstein's theorem[i] is in class $\mathfrak{C}_{U,7}^{ind,2}$ and the Collatz conjecture is in the class $\mathfrak{C}_{U,1}^{ind,2}$.

[i]Assuming the coding for Goodstein's theorem was done in a reasonable way.

This is indeed a very interesting observation. First and foremost, the complexity measure discussed here compares the hardness of solving problems *in a very particular way: by searching for a counter-example*. Goodstein's theorem can be proven rather easily in second order arithmetic, but cannot be proved in Peano arithmetic [21]: if Goodstein's theorem would be proven in Peano Arithmetic (PA) then one would be able to prove the consistency of PA, which is known to be unprovable from within PA. It follows that Goodstein's theorem appears to be more complicated than the Collatz conjecture because there is no (strong) evidence that the conjecture might be also unprovable in PA.

We also note that it is possible that a problem is more complex than another problem in the sense of our measure, yet the mathematical practice indicates that the contrary relation might be true (because different methods have been used to solve the problem). The last situation is as "provisional" as our complexity estimation, as new, possible shorter proofs, can always be discovered.

There are many open problems. In this paper, as well as in [9, 6, 7, 8], only problems formulated in the language of the first order logic have been considered, and mainly number-theoretical. It will be interesting to study complexity of mathematical problems formulated in the language of the second order logic. What is the highest complexity order of well-known mathematical problems is another open question.

Acknowledgement

We thank the referees for excellent comments which improved the paper.

References

[1] M. Burgin. *Super-recursive Algorithms*, Springer, Heidelberg, 2005.
[2] M. Burgin. Superrecursive hierarchies of algorithmic problems, in *Proceedings of the 2005 International Conference on Foundations of Computer Science*, CSREA Press, Las Vegas, 2005, 31–37.
[3] M. Burgin. Algorithmic complexity of computational problems, *International Journal of Computing & Information Technology* 2,1 (2010), 149–187
[4] M. Burgin. *Measuring Power of Algorithms, Computer Programs, and Information Automata*, Nova Science Publishers, New York, 2010.
[5] C. S. Calude. *Information and Randomness: An Algorithmic Perspective*, 2nd Edition, Revised and Extended, Springer-Verlag, Berlin, 2002.
[6] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 1 *Complex Systems*, 18-3 (2009), 267–285.
[7] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 2 *Complex Systems*, 18-4, (2010), 387–401.
[8] C. S. Calude, E. Calude. The complexity of the four colour theorem, *LMS J. Comput. Math.* 13 (2010), 414–425.
[9] C. S. Calude, E. Calude, M. J. Dinneen. A new measure of the difficulty of problems, *Journal for Multiple-Valued Logic and Soft Computing* 12 (2006), 285–307.
[10] C. S. Calude, E. Calude, M. S. Queen. The complexity of Euler's integer partition theorem, *Theoretical Computer Science* 54 (2012), 72–80.
[11] C. S. Calude, E. Calude, M. S. Queen. Inductive complexity of P versus NP problem. Extended abstract, in J. Durand-Lose, N. Jonoska (eds.). *Proceedings UCNC 2012*, LNCS 7445, Springer, (2012), 2–9.
[12] C. S. Calude, E. Calude, K. Svozil. The complexity of proving chaoticity and the Church-Turing Thesis, *Chaos* 20 037103 (2010), 1–5.
[13] C. S. Calude, M. J. Dinneen. Exact approximations of omega numbers, *Int. Journal of Bifurcation & Chaos* 17, 6 (2007), 1937–1954.
[14] C. S. Calude, M. J. Dinneen, C.-K. Shu. Computing a glimpse of randomness, *Experimental Mathematics* 11, 2 (2002), 369–378.
[15] E. Calude. The complexity of Riemann's Hypothesis, *Journal for Multiple-Valued Logic and Soft Computing*, 17, 4 (2011) 1–9.

[16] E. Calude. Fermat's Last Theorem and chaoticity, *Natural Computing*, (2011), DOI: 10.1007/s11047-011-9282-9.
[17] J. P. Davalan. $3x + 1$, Collatz, Syracuse problem, `http://pagesperso- orange.fr/jean-paul.davalan/liens/liens_syracuse.html`, (accessed on 30 November 2012).
[18] M. J. Dinneen. A program-size complexity measure for mathematical problems and conjectures, in M. J. Dinneen, B. Khoussainov, A. Nies (eds.). *Computation, Physics and Beyond*, LNCS 7160, Springer, Heidelberg, 2012, 81–93.
[19] R. K. Guy. Problem E16 in *Unsolved Problems in Number Theory*. Springer, New York, 2004, 330–336 (3rd edition).
[20] J. Hertel. Inductive complexity of Goldstein's theorem, in J. Durand-Lose, N. Jonoska (eds.). *Proceedings 11th International Conference "Unconventional Computation and Natural Computation"*, LNCS 7445, Springer, 2012, 141–151.
[21] L. Kirby, J. Paris. Accessible independence results for Peano arithmetic, *Bulletin of the London Mathematical Society* 14 (1982), 285–293.
[22] J. Lagarias. The $3x + 1$ problem and its generalizations, *Amer. Math. Monthly* 92 (1985), 3–23.
[23] J. Lagarias (ed.), *The Ultimate Challenge: The* $3x + 1$ *Problem*, AMS, 2010.
[24] R. Soare. *Recursively Enumerable Sets and Degrees*, Springer-Verlag, Berlin, 1987.
[25] 2000 Mathematics Subject Classification, MSC2000, `http://www.ams.org/msc`; see also Mathematics on the Web, `http://www.mathontheweb.org/mathweb/mi-classifications.html`, (accessed on 30 November 2012).