Algorithmic Complexity of Mathematical Problems: An Overview of Results and Open Problems

Cristian S. Calude
Department of Computer Science
The University of Auckland, New Zealand
www.cs.auckland.ac.nz/~cristian

Elena Calude
Institute of Natural and Mathematical Sciences
Massey University at Auckland, New Zealand
http://www.massey.ac.nz/~ecalude

November 25, 2012

Abstract

This paper reviews current progress and open problems in algorithmically evaluating the complexity of a large class of mathematical problems/conjectures/statements.

1 Introduction

Evaluating (or even guessing) the degree of complexity of an open problem, conjecture or mathematically proven statement is notoriously hard not only for beginners, but also for the most experienced mathematicians.

Is there a (uniform) method to evaluate, in some objective way, the difficulty of a mathematical statement or problem? 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 [24]. But, is there any indication that all, or most, or even a large part of mathematical problems have a kind of "commonality" allowing a uniform evaluation of their complexity? How could one compare a problem in number theory with a problem in complex analysis, a problem in algebraic topology or a theorem in dynamical systems?

Surprisingly enough, such "commonalities" do exist for many mathematical problems. One of them is based on the possibility of expressing the problem in terms of (very) simple programs reducible to a (natural) question in theoretical computer science, the so-called *halting problem* [7]. A more general "commonality" can be discovered using the inductive type of computation [1]. As a consequence, uniform approaches for evaluating the complexity of a large class of mathematical problems/conjectures/statements can be, and have been, developed. This paper reviews current progress and some open problems.

2 What do these mathematical sentences have in common?

Consider the following mathematical statements: 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 Collatz conjecture, the P=NP problem. What do they have in common? Apparently very little, apart from their notoriety. Some are (famous) conjectures, some are well-known theorems. They belong to different areas of mathematics, number theory, Diophantine equations, complex analysis, graph theory. Among theorems, some state positive facts, some negative results; some are very old, some are relatively new.

A mathematical "common" property of all statements above is their "logical form": they can be represented as Π_1-statements, i.e. statements of the form "$\forall n\, P(n)$", where P is a computable predicate. For some statements, like the Goldbach's conjecture and Fermat's last theorem, this is obvious; for others, like for the Riemann hypothesis, this is not trivial to prove.

The "commonality" discussed above shows that all the above sentences are *finitely refutable*: a single counter-example proves the statement false. This is the base for the development of the first method of evaluating the complexity of mathematical sentences. This method applies to all Π_1-statements, a large class of mathematical sentences, but, obviously, not all (see [13] for more mathematical facts). Simple examples for which the method does not apply are the twin prime conjecture—there are infinitely many primes p such that $p+2$ is also prime—believed to be true because of the probabilistic distribution of primes, and the conjecture that there are infinitely many Mersenne primes[1], believed to be true because the harmonic series diverges.

Intuitively, the complexity of a Π_1–problem $\pi = \forall n\, P(n)$ is measured by the size of the "smallest/simplest" program which systematically searches for a counter-example for π: if $P(0)$ is false then check $P(1)$; if $P(1)$ is false then check $P(2)$; if $P(2)$ is false then check $P(3)$, and so on. This program semi-decides the problem π because the program stops if and only if there exists an m such that $P(m)$ is false; in particular, if π is false then the program never stops. Accordingly, the resulting complexity is a semi-decidability complexity measure.

The semi-decidability complexity C_U—see the formula (1) for a precise definition—evaluates the complexity of the most intuitive way to solve a problem, a brute-force search for a counter-example. If the conjecture is false, a counter-example will eventually be found. But if a conjecture is true, the search will run on forever. 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. The search for a counter-example should be "genuine" even in case the problem is solved in the affirmative or negative; in this way we exclude the use of trivial predicates P, e.g. a constant predicate. The method is not based on the subjective expertise of a given expert but on the minimal amount of knowledge necessary to solve "in principle" the problem. Last but not least, the predicate P has to be effectively and explicitly obtained from the problem.[2]

In this way we can uniformly and objectively compare problems from different areas of mathematics, a task generally impossible for human experts, because there are few mathematicians with such expertise. However, the comparison is limited by the uniformity of the solution we analyse—we evaluate only one possible solution from infinitely

[1] I.e. numbers of the form $2^n - 1$.

[2] The importance of this requirement will be seen in the discussion of the Collatz conjecture.

many candidates. In particular, knowing the complexity of an open Π_1–problem does not give any clue about possible ways to solve the problem.

A generalisation of this method, which uses a more powerful type of computation—the inductive computation—evaluates the complexity of the decidability of problems and can be applied to the larger class of Π_2–statements. We illustrate it for the Collatz conjecture, the twin prime conjecture and the P=NP problem.

Finally, a few open questions are discussed.

3 The semi-decidability complexity measure

The intuitive approach described at the end of the previous section can be mathematically modelled in the following way. First we fix a formalism describing a universal prefix-free Turing machine[3] U; in our case we will use register machine programs, cf. [7, 4, 5]. The machine U (which is fully described in [5]) 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 $\pi = \forall m P(m)$ we associate the algorithm $\Pi_P = \inf\{n : P(n) = \text{ false}\}$. There are many programs (for U) which implement Π_P; without loss of generality, any such program will be denoted also by Π_P.

There are computable predicates P' such that $\pi = \forall m P(m) = \forall m P'(m)$, but some of them 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 this will not reflect any "algorithmic knowledge" about π permitting a "genuine" search for a counter-example, say by a proof-assistant; such a search 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, hence, on purpose it will be kept informal.

The complexity (with respect to U) of a Π_1–problem π is defined by the length of the smallest-length program (for U) Π_P—defined as above—where minimisation is calculated for all possible "genuine" representations $\pi = \forall n P(n)$ and all implementations in U of the program Π_P:[4]

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

Note that π is true if and only if $U(\Pi_P)$ never halts.

Another way of defining the 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 [3]) 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.

If instead of U we use a different universal self-delimiting Turing machine U' then one can compute a constant c (depending upon U and U') such that for every Π_1–problem

[3]The domain of the machine is prefix-free, i.e. no proper prefix of a string in the domain is included in the domain.

[4]For C_U it is irrelevant whether π is known to be true or false.

π one has

$$|C_U(\pi) - C_{U'}(\pi)| \leq c. \tag{2}$$

Experimental calculation shows that for minimal machines the constant c is smaller than 2^{10}.

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 [5] we classify Π_1–problems into the following classes:

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

The adopted threshold 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.

4 A concrete universal prefix-free binary Turing machine

Let π, ρ be Π_1–problems. If U and U' are universal self-delimiting Turing machines and $C_U(\pi) \leq C_U(\rho)$, then $C_{U'}(\pi) \leq C_{U'}(\rho) + 2c$, where c comes from (2). For this reason in what follows we fix a universal self-delimiting Turing machine and evaluate the complexity with respect to this machine.

We briefly describe the syntax and the semantics of a register machine language which implements a (natural) minimal universal prefix-free binary Turing machine U; it is a refinement, constructed in [5], of the languages in [12, 7].

Any register program (machine) uses a finite number of registers, each of which may contain an arbitrarily large non-negative integer.

By default, all registers, named with a string of lower or upper case letters, are initialised to 0. Instructions are labelled by default with 0,1,2, aso.

The register machine instructions are listed below. Note that in all cases R2 and R3 denote either a register or a non-negative integer, while R1 must be a register. When referring to R we use, depending upon the context, either the name of register R or the non-negative integer stored in R.

=R1,R2,R3
If the contents 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. If the content of R3 is greater than the number of instructions in the program, then we have an illegal-branch error.

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

+R1,R2
The contents 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 contents of R1 becomes either 0 or 1. Any attempt to read past the last data-bit results in an over-read error. The read instruction is necessary for the universality of the register machine language; it will not be used in our programs.

%
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* consists of a finite list of labelled instructions from the above list, with the restriction that the halt instruction appears only once, as the last instruction of the list. The input data (a binary string) follows immediately after the halt instruction. A program not reading the whole data or attempting to read past the last data-bit results in a run-time error. Some programs (as the ones presented in this paper) have no input data; these programs cannot halt with an under-read error.

The instruction `=R,R,n` is used for the unconditional jump to the n-th instruction of the program. For Boolean data types we use integers 0 = `false` and 1 = `true`.

For longer programs it is convenient to distinguish between the main program and some sets of instructions called "routines" which perform specific tasks for another routine or the main program. The call and call-back of a routine are executed with unconditional jumps.

5 Binary coding of programs

In this section we develop a systematic efficient method to uniquely code in binary the register machine programs. To this aim we use a prefix-free coding (the set of code-words is prefix-free); decoding is unique and instantaneous.

The binary coding of special characters (instructions and comma) is the following (ε is the empty string):

Table 1. Special characters

special characters	code	special characters	code
,	ε	+	111
&	01	!	110
=	00	%	100

For registers we use the prefix-free code $\text{code}_1 = \{0^{|x|}1x \mid x \in \{0,1\}^*\}$. Here are the codes of the first 32 registers:[5]

Table 2. Registers

register	code_1	register	code_1
R_1	010	R_9	0001010
R_2	011	R_{10}	0001011
R_3	00100	R_{11}	0001100
R_4	00101	R_{12}	0001101
R_5	00110	R_{13}	0001110
R_6	00111	R_{14}	0001111
R_7	0001000	R_{15}	000010000
R_8	0001001	R_{16}	000010001

[5]The register names are chosen to optimise the length of the program, i.e. the most frequently used registers have the smallest code_1 length.

Table 2. Registers (cont.)

R_{17}	000010010	R_{25}	000011010
R_{18}	000010011	R_{26}	000011011
R_{19}	000010100	R_{27}	000011100
R_{20}	000010101	R_{28}	000011101
R_{21}	000010110	R_{29}	000011110
R_{22}	000010111	R_{30}	000011111
R_{23}	000011000	R_{31}	00000100000
R_{24}	000011001	R_{32}	00000100001

For non-negative integers we use the prefix-free code $\text{code}_2 = \{1^{|x|}0x \mid x \in \{0,1\}^*\}$. Here are the codes of the first 16 non-negative integers:

Table 3. Non-negative integers

integer	code_2	integer	code_2
0	100	4	11010
1	101	5	11011
2	11000	6	1110000
3	11001	7	1110001
8	1110010	12	1110110
9	1110011	13	1110111
10	1110100	14	111100000
11	1110101	15	111100001

The instructions are coded by self-delimiting binary strings as follows:

1. `&R1,R2` is coded in two different ways depending on R2:[6]

$$01\text{code}_1(\text{R1})\text{code}_i(\text{R2}),$$

   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.

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

$$100.$$

[6] As $x\varepsilon = \varepsilon x = x$, for every string $x \in \{0,1\}^*$, in what follows we omit ε.

All codings for instruction names, special symbol comma, registers and non-negative integers are self-delimiting; 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.

The smallest program which halts is 100 and smallest program which never halts is 00010010100100.

6 Programming techniques

A very important tool for coding sequences is Cantor's bijection which maps (codes) a pair of non-negative integers a, b into a single non-negative integer $\langle a, b\rangle$. This function can be iterated to a bijection between $\mathbf{N}^k$ and $\mathbf{N}$, for every $k > 1$; we shall adopt, by convention, the left-associative iteration. For example, to work with arrays in register machine programs we need to code (finite) sequences of non-negative integers into single non-negative integers.

For example the 4-element sequence $[2, 1, 1, 0]$ is encoded by 1484 as $\langle\langle\langle 2, 1\rangle, 1\rangle, 0\rangle = \langle\langle 8, 1\rangle, 0\rangle = \langle 53, 0\rangle = 1484$. The reverse process allows to convert, for each given $k \geq 1$, any non-negative integer into a unique k-element sequence of non-negative integers. For example, the integer 5564 can be converted to the 2-element sequence $[104, 0]$ and the 4-element sequence $[3, 1, 0, 0]$ via the decomposition $5564 = \langle 104, 0\rangle = \langle\langle 13, 0\rangle, 0\rangle = \langle\langle\langle 3, 1\rangle, 0\rangle, 0\rangle$.

Cantor's bijection is efficient for codings of large data; the coding presented [16] is preferable for smaller data. The routine below computes the Cantor's function using the formula $\langle a, b\rangle = 1 + 2 + \ldots + (a + b) + a$.

Table 4: Cantor's bijection with 38 instructions

nr	label	instruction
0		`=a, a, CAN`
1	MUL	`&d, 0`
2		`&e, 0`
3	LM1	`=e, b, c`
4		`+d, a`
5		`+e, 1`
6		`=a, a, LM1`
7	DIV	`&d, 1`
8		`&e, b`
9		`&f, 0`
10	LD1	`=e, a, c`
11		`+e, 1`
12		`+f, 1`
13		`=f, b, LD3`
14		`=a, a, LD1`
15	LD3	`&f, 0`
16		`+d, 1`
17		`=a, a, LD1`
18	CAN	`&ac, a`
19		`&bc, b`
20		`&cc, c`

Table 4: Cantor's bijection with 38 instructions (cont.)

nr	label	instruction
21		`&d, a`
22		`+d, b`
23		`&ec, d`
24		`+ec, 1`
25		`&a, d`
26		`&b, ec`
27		`&c, LC1`
28		`=a, a, MUL`
29	LC1	`&a, d`
30		`&b, 2`
31		`&c, LC2`
32		`=a, a, DIV`
33	LC2	`+d, a`
34		`&a, ac`
35		`&b, bc`
36		`&c, cc`
37		`=a, a, c`

This routine has 38 instructions and the program size is 480 bits. The routine includes two stand-alone routines, one for the multiplication `d=a*b`, encoded in the instructions 1 to 6, and the other one for integer division `d=[a/b],b>0`, encoded in the instructions 7 to 17. Note that the value of `c`, used inside a routine to get back to the calling environment, is set—to a non-zero integer value—before the routine is called, therefore it creates no infinite loop.

Mathematical elegance is not always a guarantee for program-size optimality. For example, we can compute Cantor's bijection with the shorter program, presented in Table 5, using the mathematically "ugly" formula $\langle a, b\rangle = 1 + 2 + \ldots + (a + b) + a$:

Table 5: Cantor's bijection with 11 instructions

nr	instruction	code	length
0	`&e, a`	01 011 010	8
1	`+e, b`	100 011 00110	11
2	`&d, 0`	01 00101 100	10
3	`=e, 0, 9`	101 011 100 1110011	16
4	`&f, 1`	01 00100 101	10
5	`+d, f`	100 00101 00100	13
6	`=e, f, 9`	101 011 00100 1110011	18
7	`+f, 1`	100 00100 101	11
8	`=a, a, 5`	101 010 010 11011	14
9	`+d, a`	100 00101 010	11
10	`=a, a, c`	101 010 010 00111	14

This register machine program has 136 bits:

01011010100011001100100101100101011100111001101001001011000010100100
10101100100111001110000100101101010010110111000010101010101001000111

As the first approach uses two other routines, multiplication and division, one can argue that their encodings are part of the general program therefore they should not be

counted towards the size of Cantor's function encoding. Even if we do not consider the encodings for multiplication (6 instructions) and division (11 instructions) as part of the Cantor's encoding in Table 4, the number of instructions (21) is still larger than in the approach presented in Table 5 (11 instructions).

7 Some results

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 of a power of two is never a power of five) is in $\mathfrak{C}_{U,2}$ [4, 5, 18, 15], the Riemann hypothesis (all non-trivial zeros of the Riemann zeta function have real part 1/2) and Euler's 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 $\mathfrak{C}_{U,3}$ [14, 8], 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 $\mathfrak{C}_{U,4}$ [6]. Related results have been discussed in [10].

8 The Collatz conjecture

The Collatz conjecture[7] 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

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

There is a huge literature on this problem and various natural generalisations: see [21]. Does there exist a program Π_{Collatz} such that Collatz's conjecture is false if and only if Π_{Collatz} halts? 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 these cases? How can one refute b) by a brute-force tester? We do not know the answers to the above questions. However, a simple non-constructive argument [7] answers in the affirmative the first question of this section. Indeed, observe first that the set

$$Collatz = \{a_1 \mid a_N = 1, \text{ for some } N > 1\}$$

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. As a) is ruled out, in this case any program which eventually halts can be taken *in principle* as Π_{Collatz}; this is not the case for most of the programs because they do not "genuinely" search for a counter-example to the Collatz conjecture.

If $Collatz$ is computable, then we can write a program Π_{Collatz} using the computable predicate defining $Collatz$: the conjecture is true if and only if Π_{Collatz} never stops.

The above observation shows that, in principle, the method developed can be applied for the Collatz conjecture. In fact, the method cannot be applied, at least for the time

[7] Known as Collatz's conjecture, the $3x + 1$ problem, or Ulam's problem.

being, as we do not know how to explicitly construct the program Π_{Collatz}. This raises the question of extending the method such that it can be applied to the Collatz conjecture and similar mathematical statements.

It is easy to see that the Collatz conjecture is a Π_2–statement, i.e. a statement of the form $\forall n \exists i R(n,i)$, for some computable binary predicated R. The above non-constructive proof given for the Collatz conjecture works for every Π_2–statement. Obviously, for infinitely many Π_2–statements this proof cannot be constructivised, that is, infinitely many Π_2–statements are not provably Π_1–statements.

9 A general method: Inductive complexity measures

Can the complexity method developed for Π_1–statements be extended to Π_2–statements? The algorithmic brute force verification of the validity of a Π_2–statement does not work as for the simpler Π_1–statements: the program never stops irrespective whether the statement is true or false. A natural solution is to use inductive Turing computations [1] (which are $\mathbf{0}'$–computations by Shöenfield's lemma) instead of classical Turing computations.

We recall following [1] that an inductive Turing machine of the first order is a normal Turing machine with input, output and working tapes, which computes "inductively": The result of the computation of such an inductive Turing machine M on 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. So, in contrast with the (classical) computation of the Turing machine M on x—which assumes that the computation has stopped and the result is the content of the output tape—the inductive computation of M on x may stop and in this case the result is the same as in the classical mode, or may not stop, in which case there is a result only when the content of the output tape has stabilised at some step of the (infinite) computation. In this case, we say that M is an inductive Turing machine of first order.

First we note that the method of evaluating the complexity of Π_1–statements can be reformulated in terms of inductive Turing machines of first order. To the computable predicate $P(m)$ we assign the problem $\pi = \forall m P(m)$, the algorithm $\Pi_P = \inf\{n : P(n) = \text{ false}\}$ and finally the inductive program of first order $\Pi_P^{ind,1}$ constructed from the program Π_P in which the stop instruction, %, is replaced with the instruction `&a, 1` followed by %; here `a` is a register not appearing in Π_P designed as the output register. Denote by U^{ind} the machine U working inductively. It is easy to see that

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

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

$$C_U^{ind,1}(\pi) = \min\{|\Pi_P^{ind,1}| : \pi = \forall n P(n)\}, \tag{4}$$

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

$$\mathfrak{C}_{U,n}^{ind,1} = \{\pi : \pi \text{ is a } \Pi_1\text{–statement}, C_U^{ind,1}(\pi) \leq 2^{10} n\}. \tag{5}$$

There is a constant $c < 1024$ such that for every Π_1–statement π we have:

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

so

$$\mathfrak{C}_{U,n} \approx \mathfrak{C}_{U,n}^{ind,1}.$$

More precisely,

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

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? Because using U^{ind} we can extend the first method from sentences $\forall m P(m)$ to more complex sentences, in particular, to Π_2–sentences.

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

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

Next we construct the inductive Turing machine $M_R^{ind,2}$ defined by

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

Clearly,

$$M_R^{ind,2} = \begin{cases} 0, & \text{if } \forall n\, (T_E^{ind,1}(n) = 1), \\ 1, & \text{otherwise}, \end{cases}$$

hence we say that $M_R^{ind,2}$ is an *inductive Turing machine of second order*.

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:

$$\forall n \exists i R(n,i) \text{ is true if and only if } U^{ind}(M_R^{ind,2}) = 0.$$

In this way, the inductive complexity measure of first order $C_U^{ind,1}(\pi)$ (see (4)) 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 (5)) to the *inductive complexity class of second order*:

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

Using this technique, the Collatz conjecture and the twin prime conjecture are in the inductive complexity class $\mathfrak{C}_{U,3}^{ind,1}$ (see [2]), Goodstein's theorem (every "Goodstein sequence" eventually terminates at 0; the theorem is unprovable in Peano arithmetic [20]) is in $\mathfrak{C}_{U,2}^{ind,7}$ (see [17]), and the P=NP problem is in $\mathfrak{C}_{U,3}^{ind,7}$, cf. [9].

10 Open problems

In this section we present a few open problems.

1. Find methods to prove lower bounds for the complexity of problems and use them for the problems studied.

2. Can one construct programs Π_{Collatz} and $\Pi_{\text{twinprimeconjecture}}$, i.e. are the statements of the Collatz and twin prime conjectures provably Π_1–statements?

3. Evaluate the complexity of the Poincaré theorem [23]; see also [19, 22].

4. (Dinneen [16]) Consider the universal machine U presented in section 6 (see [5]). Can one determine how many initial bits of its halting probability (if any) can be computed? Compare this result with the 40 bits that were computed in [12, 11] to get some indirect information regarding of how far away we are from actually solving mathematical conjectures such as the Riemann's hypothesis. See other related open problems in [16].

11 Conclusions

A uniform method for evaluating the complexity of mathematical problems represented by Π_1–statements was described. The method was applied to a variety of problems, including the Fermat last theorem, the Goldbach conjecture, the four colour problem and the Riemann hypothesis. The complexity of some problems in this class, like the Collatz conjecture and the twin prime conjecture, has not been evaluated because we could not explicitly construct the computable predicate appearing in the corresponding Π_1–statements. Hence, the method was extended, using inductive Turing machine computations instead of Turing machine computations, to a larger class of problems including all Π_2–statements. In this way the inductive complexity of the Collatz, twin prime conjectures and P=NP problem have been evaluated.

The scalability of the measure, both in terms of ordering, the role of the additive constants involved, and its relativisation to various unsolvable problems are open questions. Further interesting open problems are discussed in the previous section.

Acknowledgement

We thank Mark Burgin, Michael Dinneen, Yun-Bum Kim and Sergiu Rudeanu for discussions on the topic of this paper and suggestions that improved the presentation.

References

[1] M. Burgin. *Super-recursive Algorithms*, Springer, Heidelberg, 2005.

[2] M. Burgin, C. S. Calude, E. Calude. Inductive Complexity Measures for Mathematical Problems, *CDMTCS Research Report* 416, 2011, 11 pp.

[3] C. S. Calude. *Information and Randomness: An Algorithmic Perspective*, Springer, Berlin, 2002. (2nd ed.)

[4] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 1 *Complex Systems*, 18-3 (2009), 267–285.

[5] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 2 *Complex Systems*, 18-4, (2010), 387–401.

[6] C. S. Calude, E. Calude. The complexity of the Four Colour Theorem, *LMS J. Comput. Math.* 13 (2010), 414–425.

[7] 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.

[8] C. S. Calude, E. Calude, M. S. Queen. The complexity of Euler's integer partition theorem, *Theoretical Computer Science* 454 (2012), 72–80.

[9] 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 11th International Conference "Unconventional Computation and Natural Computation"*, LNCS 7445, Springer, (2012), 2–9.

[10] C. S. Calude, E. Calude, K. Svozil. The complexity of proving chaoticity and the Church-Turing Thesis, *Chaos* 20 037103 (2010), 1–5.

[11] C. S. Calude, M. J. Dinneen. Exact approximations of omega numbers, *Int. Journal of Bifurcation & Chaos* 17, 6 (2007), 1937–1954.

[12] C. S. Calude, M. J. Dinneen, C.-K. Shu. Computing a glimpse of randomness, *Experimental Mathematics* 11, 2 (2002), 369–378.

[13] C. S. Calude, H. Jürgensen, S. Legg. Solving finitely refutable mathematical problems, in C. S. Calude, G. Păun (eds.). *Finite Versus Infinite. Contributions to an Eternal Dilemma*, Springer-Verlag, London, 2000, 39–52.

[14] E. Calude. The complexity of Riemann's Hypothesis, *Journal for Multiple-Valued Logic and Soft Computing*, 18 (3-4) (2012), 257–265.

[15] E. Calude. Fermat's Last Theorem and chaoticity, *Natural Computing*, (2011), DOI: 10.1007/s11047-011-9282-9.

[16] 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.

[17] J. Hertel. Inductive complexity of Goodstein's theorem, in J. Durand-Lose, N. Jonoska (eds.). *Proceedings 11th International Conference "Unconventional Computation and Natural Computation"*, LNCS 7445, Springer, 2012, 141–151.

[18] J. Hertel. On the Difficulty of Goldbach and Dyson Conjectures, *CDMTCS Research Report* 367, 2009, 15pp.

[19] M. Kim. *Why Everyone Should Know Number Theory*, Manuscript `http://www.ucl.ac.uk/~ucahmki/numbers.pdf`, 1998 (accessed on 10 October 2011).

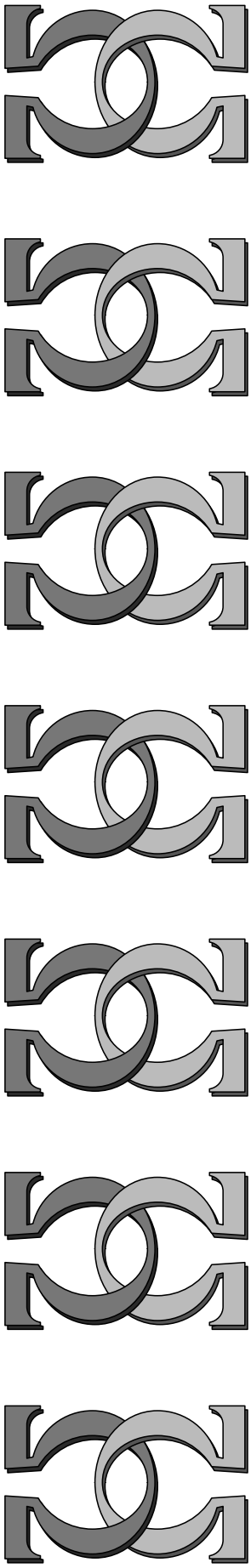
[20] L. Kirby, J. Paris. Accessible independence results for Peano arithmetic, *Bulletin of the London Mathematical Society* 14 (1982), 285–293.

[21] J. Lagarias (ed.), *The Ultimate Challenge: The* $3x + 1$ *Problem*, AMS, 2010.

[22] J. Manning. Algorithmic detection and description of hyperbolic structures on closed 3–manifolds with solvable word problem, *Geometry & Topology* 6 (2002), 1–26.

[23] G. Perelman. Ricci Flow and Geometrization of Three-Manifolds, Massachusetts Institute of Technology, Department of Mathematics Simons Lecture Series, September 23, 2004 `http://www-math.mit.edu/conferences/simons` (accessed on 10 October 2011).

[24] 2000 Mathematics Subject Classification, MSC2000, `http://www.ams.org/msc`; see also Mathematics on the Web, `http://www.mathontheweb.org/mathweb/mi-classifications.html`.



CDMTCS
Research
Report
Series

The Complexity of Mathematical Problems: An Overview of Results and Open Problems

Cristian S. Calude,[1] Elena Calude[2]

[1]University of Auckland, NZ
[2]Massey University at Auckland, NZ

CDMTCS-410
November 2011

Centre for Discrete Mathematics and
Theoretical Computer Science