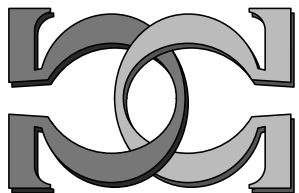
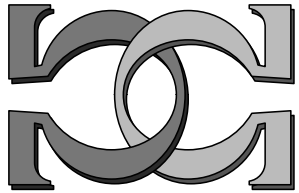
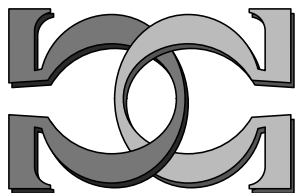


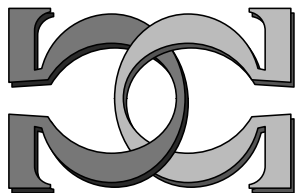
**CDMTCS  
Research  
Report  
Series**



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

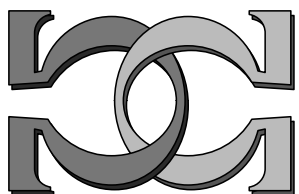


**Cristian S. Calude,<sup>1</sup> Elena Calude<sup>2</sup>**

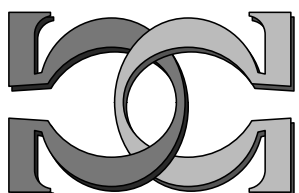


<sup>1</sup>University of Auckland, NZ

<sup>2</sup>Massey University at Auckland, NZ



CDMTCS-410  
November 2011



Centre for Discrete Mathematics and  
Theoretical Computer Science

# THE COMPLEXITY OF MATHEMATICAL PROBLEMS: AN OVERVIEW OF RESULTS AND OPEN PROBLEMS

CRISTIAN S. CALUDE AND ELENA CALUDE

ABSTRACT. This paper presents an overview of results obtained with an algorithmic uniform method to measure the complexity of a large class of mathematical problems and discusses a few open problems.

## 1. INTRODUCTION

Evaluating (or even guessing) the degree of complexity of an open problem, conjecture or mathematical 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 [23]. 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* [3]. A more general “commonality” can be discovered using the inductive type of computation proposed by Burgin [1]. As a consequence, uniform approaches for evaluating the complexity of a large class of mathematical problems/conjecture/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. What do they have in common? Apparently very little, apart 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; some are very old, some relatively new.

A mathematical “common” property of all statements above is their “form”: all can be represented as  $\Pi_1$ -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 Collatz’s conjecture is a special case: the only proof that its statement is  $\Pi_1$  is non-constructive, that is, the corresponding predicate  $P(n)$  is shown to exist, but it is not explicitly constructed<sup>1</sup>. Does it matter? In fact, we shall show that indeed, it matters!

The “commonality” discussed above shows that all the above sentences are *finitely refutable*: a single counterexample 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 a large class of mathematical sentences, but not to all (see [11] for more mathematical facts). Simple such examples are provided by the twin prime conjecture—there are infinitely many primes  $p$  such that  $p+2$  is also prime—conjecture believed to be true because of the probabilistic distribution of primes, and the conjecture that there are infinitely many Mersenne primes<sup>2</sup>, conjecture believed to be true because the harmonic series diverges.

### 3. THE FIRST COMPLEXITY MEASURE

The complexity measure [3, 4, 5] for  $\Pi_1$ -statements is defined by means of register machine programs which implement a universal self-delimiting Turing machine  $U$ . 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  $\Pi_1$ -problem  $\pi = \forall m P(m)$  we associate the algorithm  $\Pi_P = \inf\{n : P(n) = \text{false}\}$  which systematically searches for a counter-example for  $\pi$ . There are many programs (for  $U$ ) which implement  $\Pi_P$ ; without loss of generality, any such program will be denoted also by  $\Pi_P$ . Note that  $\pi$  is true iff  $U(\Pi_P)$  never halts.

The complexity (with respect to  $U$ ) of a  $\Pi_1$ -problem  $\pi$  is defined by the length of the smallest-length program (for  $U$ )  $\Pi_P$ —defined as above—where minimisation is calculated for all possible representations of  $\pi$  as  $\pi = \forall n P(n)$ :<sup>3</sup>

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

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  $\Pi_1$ -problems into the following classes:

$$\mathfrak{C}_{U,n} = \{\pi : \pi \text{ is a } \Pi_1\text{-problem, } C_U(\pi) \leq n \text{ kbit}^4\}.$$

<sup>1</sup>There is no known proof showing that a constructive proof does not exist.

<sup>2</sup>I.e. numbers of the form  $2^n - 1$ .

<sup>3</sup>For  $C_U$  it is irrelevant whether  $\pi$  is known to be true or false. In particular, the program containing the single instruction halt is not a  $\Pi_P$  program, for any  $P$ .

<sup>4</sup>A kilobit (kbit or kb) is equal to  $2^{10}$  bits.

## 4. A UNIVERSAL PREFIX-FREE BINARY TURING 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 [10, 3].

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 labeled by default with  $0,1,2,\dots$

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-th 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 outside the scope of 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 a run-time error.

**%**

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 labeled 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  $0 = \text{false}$  and  $1 = \text{true}$ .

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 as follows.

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

special characters	code	special characters	code
,	$\varepsilon$	+	111
&	01	!	110
=	00	%	100

Table 1. Special characters

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:<sup>5</sup>

register	code <sub>1</sub>	register	code <sub>1</sub>	register	code <sub>1</sub>	register	code <sub>1</sub>
R <sub>1</sub>	010	R <sub>9</sub>	0001010	R <sub>17</sub>	000010010	R <sub>25</sub>	000011010
R <sub>2</sub>	011	R <sub>10</sub>	0001011	R <sub>18</sub>	000010011	R <sub>26</sub>	000011011
R <sub>3</sub>	00100	R <sub>11</sub>	0001100	R <sub>19</sub>	000010100	R <sub>27</sub>	000011100
R <sub>4</sub>	00101	R <sub>12</sub>	0001101	R <sub>20</sub>	000010101	R <sub>28</sub>	000011101
R <sub>5</sub>	00110	R <sub>13</sub>	0001110	R <sub>21</sub>	000010110	R <sub>29</sub>	000011110
R <sub>6</sub>	00111	R <sub>14</sub>	0001111	R <sub>22</sub>	000010111	R <sub>30</sub>	000011111
R <sub>7</sub>	0001000	R <sub>15</sub>	000010000	R <sub>23</sub>	000011000	R <sub>31</sub>	00000100000
R <sub>8</sub>	0001001	R <sub>16</sub>	000010001	R <sub>24</sub>	000011001	R <sub>32</sub>	00000100001

Table 2. Registers

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:

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

Table 3. Non-negative integers

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

- (1) & R1,R2 is coded in two different ways depending on R2:<sup>6</sup>

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

where  $i = 1$  if R2 is a register and  $i = 2$  if R2 is an integer.

<sup>5</sup>The register names are chosen to optimise the length of the program, i.e. the most frequent registers have the smallest code<sub>1</sub> length.

<sup>6</sup>As  $x\varepsilon = \varepsilon x = x$ , for every string  $x \in \{0,1\}^*$ , in what follows we omit  $\varepsilon$ .

(2) + R1,R2 is coded in two different ways depending on R2:

$$11\text{code}_1(\text{R1})\text{code}_i(\text{R2}),$$

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

$$100.$$

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 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  $\langle a, b \rangle = 1 + 2 + \dots + (a+b) + a$ . 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 a single non-negative integer. In what follows we use Cantor's bijection for such codings; for a different coding see [14].

For example the 4-element sequence  $[2, 1, 1, 0]$  is encoded by 1484 as  $\langle\langle\langle 2, 1 \rangle, 1 \rangle, 0 \rangle = \langle 8, 1 \rangle, 0 \rangle = \langle 53, 0 \rangle = 1484$ . 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 number 5564 can be converted to the 4-element sequence  $[3, 1, 0, 0]$  and the 2-element sequence  $[104, 0]$ .

The routine below computes the Cantor's function using the formula  $\langle a, b \rangle = (a + b)(a + b + 1)/2 + a$ :

nr	label	instruction	nr	label	instruction	nr	label	instruction
0		=a, a, CAN	13		=f, b, LD3	26		&b, ec
1	MUL	&d, 0	14		=a, a, LD1	27		&c, LC1
2		&e, 0	15	LD3	&f, 0	28		=a, a, MUL
3	LM1	=e, b, c	16		+d, 1	29	LC1	&a, d
4		+d, a	17		=a, a, LD1	30		&b, 2
5		+e, 1	18	CAN	&ac, a	31		&c, LC2
6		=a, a, LM1	19		&bc, b	32		=a, a, DIV
7	DIV	&d, 1	20		&cc, c	33	LC2	+d, a
8		&e, b	21		&d, a	34		&a, ac
9		&f, 0	22		+d, b	35		&b, bc
10	LD1	=e, a, c	23		&ec, d	36		&c, cc
11		+e, 1	24		+ec, 1	37		=a, a, c
12		+f, 1	25		&a, d			

Table 4: Cantor’s bijection with 11 instructions

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.

We can compute Cantor’s bijection with a shorter program, presented in Table 5, using the mathematically “ugly” formula  $\langle a, b \rangle = 1 + 2 + \dots + (a + b) + a$ :

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

Table 5: Cantor’s bijection with 11 instructions

This register machine program has 136 bits:

```
01011010100011001100100101100101011100111001101001001011000010100
100101011001001110011100001001011010100101101110000101010101001
000111
```

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 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, 15, 13], the Riemann hypothesis (all non-trivial zeros of the Riemann zeta function have real part  $1/2$ ) is in  $\mathfrak{C}_{U,3}$  [12], 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], and (surprisingly) the integer partition theorem (the number of partitions of an integer into odd integers is equal to the number of partitions into distinct integers) is in  $\mathfrak{C}_{U,7}$  [7].<sup>7</sup> Related results have appeared in [8].

## 8. THE COLLATZ CONJECTURE

The Collatz conjecture<sup>8</sup> 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 [17]. Does there exist a program  $\Pi_{\text{Collatz}}$  such that Collatz's conjecture is false if and only if  $\Pi_{\text{Collatz}}$  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 don't know the answers to the above questions. However, a simple non-constructive argument [3] answers in the affirmative the first question of this section. Indeed, observe first that the set

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

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  $\Pi_{\text{Collatz}}$  as a) is ruled out. If *Collatz* is computable, then we can write a program  $\Pi_{\text{Collatz}}$  to find an integer not in *Collatz*: the conjecture is true if and only if  $\Pi_{\text{Collatz}}$  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 being, as we do not know how to explicitly construct the program  $\Pi_{\text{Collatz}}$ . This raises the question of finding a more general method which can be applied to the Collatz conjecture and similar mathematical statements.

<sup>7</sup>Mainly because of the codification of arrays.

<sup>8</sup>Known as Collatz's conjecture, the Syracuse conjecture, the  $3x + 1$  problem, Kakutani's problem, Hasse algorithm, or Ulam's problem.

## 9. A MORE GENERAL METHOD: INDUCTIVE COMPLEXITY MEASURES

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 in the case 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.

The method of evaluating the problem complexity can be reformulated in terms of inductive Turing machines of first order. To the sentence (predicate)  $\forall m P(m)$ , we assign the problem  $\pi = \{\forall m (P(m) \text{ is true})\}$  and to the problem  $\pi$  we associate the algorithm  $\Pi_P = \inf\{n : P(n) = \text{false}\}$  and the inductive program of first order  $\Pi_P^{ind,1}$  constructed from the program  $\Pi_P$  in which the stop instruction  $\%$  is replaced with the instruction  $\& \mathbf{a}, 1$  followed by  $\%$ ; here  $\mathbf{a}$  is a register not appearing in  $\Pi_P$  designed as 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

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

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

$$(2) \quad \mathfrak{C}_{U,n}^{ind,1} = \{\pi : \pi \text{ is a } \Pi_1\text{-statement, } C_U^{ind,1}(\pi) \leq n \text{ kbit}\}.$$

There is a (small) constant  $c$  such that for every  $\Pi_1$ -statement  $\pi$  we have:

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

so

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

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, for sentences of the form  $\forall n \exists i R(n, i)$ , where  $R$  is a computable binary predicate.

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  $M_R^{ind,2}$  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 (1)) can be extended to the:

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

and the inductive complexity class of first order  $\mathfrak{C}_{U,n}^{ind,1}$  (see (2)) to the *inductive complexity class of second order*:

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

Using this technique, in [2] one proves that the Collatz conjecture and the twin prime conjecture are in the inductive complexity class  $\mathfrak{C}_{U,3}^{ind,2}$ .

## 10. OPEN PROBLEMS

In this section we present a few open problems arising naturally.

- (1) For every  $n$  there exists  $m > n$  such that  $\mathfrak{C}_{U,n} \subset \mathfrak{C}_{U,m}$ . Can we take  $m = n + 1$ ? The same problem for the classes  $C_{U,n}^{ind,k}$ .
- (2) Find methods to prove lower bounds for the complexity of problems and use them for the problems studied.
- (3) Can one construct programs  $\Pi_{\text{Collatz}}$  and  $\Pi_{\text{twinprimeconjecture}}$ ?
- (4) Evaluate the complexity of the Poincaré conjecture [20, 21, 22]. (Note that Kim [16] discusses the possibility that the Poincaré conjecture, is equivalent to the unsolvability of a Diophantine equation (see also [18]). If this would be true then our method would offer, at least in principle, an indication of the difficulty of this problem too.)
- (5) Evaluate the complexity of the P=?NP problem (see [19]).

- (6) (Dinneen [14]) Develop an optimising compiler that could produce machine-level register machines from the higher-level human-level specifications. It should support different encodings and allow the users to compare their programs complexity (bit-length sizes) under different models.
- (7) (Dinneen [14]) Consider the universal machine  $U$  presented in section 6 (see [5]). Can one determine how many initial bits of its halting probability (if any) that can be computed? Compare this result with the 40 bits that were computed for the first-generation register machine in [10, 9]. In this way we may 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 [14].
- (8) (Dinneen [14]) Implement other simple data structures like dictionaries and graphs to expand the library of core subroutines developed in [14] and [7].

## 11. CONCLUSIONS

A uniform method for evaluating the complexity of mathematical problems of the form  $\forall n P(n)$  was described. The method was applied to a variety of problems, including the Fermat last theorem, the Goldbach conjecture, the four colour problem, the Riemann hypothesis, and the Hilbert 10th problem. 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 predicate  $P$ . Hence, the method was extended, using inductive Turing machine computations instead of Turing machine computations, to the larger class of problems of the form  $\forall n \exists i R(n, i)$ . In this way the inductive complexity of the Collatz and twin prime conjectures 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 eight interesting open problems are discussed in the last section.

## ACKNOWLEDGEMENT

We thank Mark Burgin, Michael Dinneen and Eric Goles for discussions on the topic of this paper and for 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, in preparation.
- [3] 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.
- [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. The complexity of Euler's integer partition theorem, *CDMTCS Research Report* 409 (2011), 11 pp.
- [8] C. S. Calude, E. Calude and K. Svozil. The complexity of proving chaoticity and the Church-Turing Thesis, *Chaos* 20 037103 (2010), 1–5.

- [9] C. S. Calude, M. J. Dinneen. Exact approximations of omega numbers, *Int. Journal of Bifurcation & Chaos* 17, 6 (2007), 1937–1954.
- [10] C. S. Calude, M. J. Dinneen and C.-K. Shu. Computing a glimpse of randomness, *Experimental Mathematics* 11, 2 (2002), 369–378.
- [11] 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.
- [12] E. Calude. The complexity of Riemann’s Hypothesis, *Journal for Multiple-Valued Logic and Soft Computing*, (2012), to appear.
- [13] E. Calude. Fermat’s Last Theorem and chaoticity, *Natural Computing*, (2011), DOI: 10.1007/s11047-011-9282-9.
- [14] M. J. Dinneen. A program-size complexity measure for mathematical problems and conjectures, in M. J. Dinneen, B. Khossainov, A. Nies (eds.). *Computation, Physics and Beyond*, Springer, Heidelberg, 2012. (to appear)
- [15] J. Hertel. On the Difficulty of Goldbach and Dyson Conjectures, *CDMTCS Research Report* 367, 2009, 15pp.
- [16] M. Kim. *Why Everyone Should Know Number Theory*, manuscript <http://www.ucl.ac.uk/~ucahmki/numbers.pdf>, 1998 (accessed on 10 October 2011).
- [17] J. Lagarias (ed.), *The Ultimate Challenge: The  $3x + 1$  Problem*, AMS, 2010.
- [18] J. Manning. Algorithmic detection and description of hyperbolic structures on closed 3-manifolds with solvable word problem, *Geometry & Topology* 6 (2002), 1–26.
- [19] C. Moore, S. Mertens. *The Nature of Computation*, Oxford University Press, Oxford, 2011.
- [20] 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).
- [21] G. Perelman. The Entropy Formula for the Ricci Flow and Its Geometric Application, November 11, 2002, <http://www.arxiv.org/abs/math.DG/0211159> (accessed on 10 October 2011).
- [22] G. Perelman. Ricci Flow with Surgery on Three-Manifolds, March 10, 2003, <http://www.arxiv.org/abs/math.DG/0303109> (accessed on 10 October 2011).
- [23] 2000 Mathematics Subject Classification, MSC2000, <http://www.ams.org/msc>; see also Mathematics on the Web, <http://www.mathontheweb.org/mathweb/mi-classifications.html>.

DEPARTMENT OF COMPUTER SCIENCE, THE UNIVERSITY OF AUCKLAND, NEW ZEALAND,  
[WWW.CS.AUCKLAND.AC.NZ/~CRISTIAN](http://www.cs.auckland.ac.nz/~CRISTIAN).

INSTITUTE OF INFORMATION AND MATHEMATICAL SCIENCES, MASSEY UNIVERSITY AT  
AUCKLAND, NEW ZEALAND, [HTTP://WWW.MASSEY.AC.NZ/~ECALUDE](http://www.massey.ac.nz/~ECALUDE).