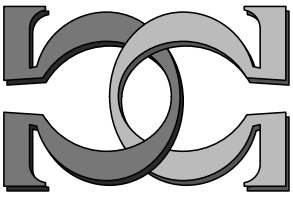

CDMTCS Research Report Series

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

INDUCTIVE COMPLEXITY MEASURES FOR MATHEMATICAL PROBLEMS

MARK BURGIN, CRISTIAN S. CALUDE AND ELENA CALUDE

Abstract. An algorithmic uniform method to measure the complexity of statements of finitely refutable statements [6, 7, 8], was used to classify famous/interesting mathematical statements like Fermat's last theorem, Hilbert's tenth problem, the four colour theorem, the Riemann hypothesis, [9, 13, 14]. 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 statements of the form $\forall n \exists m\, R(n, m)$, where $R(n, m)$ is a computable binary predicate. As illustrations, we evaluate the inductive complexity of the Collatz conjecture or twin prime conjecture—which cannot not 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 [19].

In a series of papers [6, 7, 8], 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 counterexample 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, the Collatz conjecture are examples of Π_1–statements. Not every mathematical statement is Π_1: for example, the twin prime conjecture or the conjecture that there are infinitely many Mersenne primes. The Collatz conjecture is a Π_1–statement, but no known method can explicitly produce a representation of the form $\forall n\, P(n)$ for it, hence the method, which requires the computable predicated P, cannot be effectively applied.

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

Date: January 16, 2012.

An algorithm *semi-decides* a problem ρ in case it returns 0 when ρ is false and is undefined when ρ is true. An algorithm *decides* a problem ρ in case it returns 0 when ρ is false and is 1 when ρ is true.

Accordingly, we consider the inductive complexity of the *semi-decidability* of ρ when we work with algorithms semi-deciding ρ, and the inductive complexity of the *decidability* of ρ when we work with algorithms deciding ρ. A detailed study of the decidability complexity is presented in [3].

We illustrate our method by evaluating the inductive complexity of the semi-decidability and decidability of the Collatz conjecture and twin prime conjecture.

2. A semi-decidability complexity measure

The complexity measure [6, 7, 8] for Π_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 [8]) 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 Π_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 π. There are many programs (for U) which implement Π_P; without loss of generality, any such program will be denoted also by Π_P. Note that

$$\pi \text{ is true if and only if } U(\Pi_P) \text{ never halts.}$$

The complexity (with respect to U) of a Π_1–problem π is defined by the length $|U(\Pi_P)|$ of the smallest-length program (for U) Π_P—defined as above—where minimisation is calculated for all possible representations of π as $\forall n P(n)$:[1]

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

The algorithm Π_P *semi-decides* the problem π so, $C_U(\pi)$ is a semi-decidability complexity of π.

Semi-decidability (or more exactly, positive semi-decidability) with respect to Turing machines has become a popular concept. However, there are other useful models of computation for which decidability and semi-decidability are important concepts. Decidability and semi-decidability, introduced and studied in [4] for the general context of the axiomatic theory of algorithms, encompasses virtually all existing models of computation. Note that if a problem π is false, i.e. it has a negative solution, then the complexity of its decidability is not larger than the complexity of its semi-decidability. To study the complexity of the *decidability* of π we need to go from classical algorithms to super-recursive algorithms [1].

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 [8], we classify Π_1–statements into the following classes:

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

[1] For C_U it is irrelevant whether π is known to be true or false. In particular, the program containing the single instruction halt is not a Π_P program, for any P.

[2] A kilobit (kbit or kb) is equal to 2^{10} bits.

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 of a power of two is never a power of five) is in $\mathfrak{C}_{U,2}$ [7, 8, 14], the Riemann hypothesis (all non-trivial zeros of the Riemann zeta function have real part 1/2) is in $\mathfrak{C}_{U,3}$ [13], 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 $\mathfrak{C}_{U,4}$ [9]. Related results have appeared in [10].

3. The Collatz conjecture

The Collatz conjecture[3] 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, 16, 15, 18]. Erdös is quoted by [17]) by saying that *Mathematics may not be ready for such problems.* 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 tires 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 [6] answers in the affirmative the first question of this section. Indeed, observe that the set

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

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 Π_{Collatz} as a) is ruled out. 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.

The above observation shows that although, in principle, the developed method can be applied for the Collatz conjecture, 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.

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

4. Inductive complexity measures of order k

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 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.

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 $\pi = \forall m P(m)$ and to the problem π we associate the algorithm $\Pi_P = \inf\{n : P(n) = \text{ false}\}$. The inductive program of first order $\Pi_P^{ind,1}$ is 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 output register. Denote by U^{ind} the machine U working inductively. It is easy to see that

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

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

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

There is a (small) constant c such that for every Π_1–statement π we have:

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

so

$$\mathfrak{C}_{U,n} = \mathfrak{C}_{U,n}^{ind,1}. \tag{2}$$

Note that C_U and $\mathfrak{C}_{U,n}$ refer to the semi-decidability of the problem π, while $C_U^{ind,1}$ and $\mathfrak{C}_{U,n}^{ind,1}$ refer to the decidability of the problem π. In other words, the semi-decidability complexity measure C_U was reformulated in terms of a decidability complexity measure $C_U^{ind,1}$: these two measures determine identical complexity classes, cf. (2). A direct argument for this fact will appear in Theorem 2.

There is one more reason to compute inductively instead of classically: using U^{ind} we can extend the first method from sentences $\forall m P(m)$ to more complex sentences, in particular, to 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 (1)) to the *inductive complexity class of second order*:

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

Based on the above construction and results in [2] one get the following theorem:

Theorem 1. *The decidability of every problem $\rho = \forall n \exists i R(n,i)$ belongs to an inductive complexity class of second order. The decidability of some problems ρ does not belong to any inductive complexity class of first order.*

In a natural way we can define inductive Turing machines of order $k > 2$ [1], hence the *inductive complexity measure of order k* and the *inductive complexity class of order k*.

If a problem $\rho = \forall n \exists i R(n,i)$ has a negative solution, then there is an inductive Turing machine of the first order that solves this problem. Indeed, the existence of a negative solution means that the predicate $\forall n \exists i R(n,i)$ is false. This condition can be checked by the following inductive Turing machine—which uses a classical Turing machine R to check the predicate $R(n,i)$ and the output register a—described below:

```
1. k=1
2. i=1
3. If R(k, i) is false, a=1 and go to 4; else a=0 and go to 6.
4. i=i+1
5. Go to 3
6. k=k+1
7. Go to 2
```

This construction proves the following:

Theorem 2. *The inductive semi-decidability of every problem* $\rho = \forall n \exists i R(n, i)$ *belongs to an inductive complexity class of first order.*

5. 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 [8].

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,...

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 = `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.

Selecting one or several registers as output registers, register machine programs can compute not only in the classical (recursive) mode, when the program halts to get a result, but also in the inductive mode described above. 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 when working in classical (recursive) mode.

Using 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. To this aim we use a prefix-free coding as follows.

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

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

Table 1. Special characters

For registers we use the prefix-free code $\text{code}_1 = \{0^{|x|}1x \mid x \in \{0,1\}^*\}$. For example, the code of R_1 is 010, and the code of R_{32} is 00000100001. In programs the register names are chosen to optimise the length of the program, i.e. the most frequent registers have the smallest code_1 length.

For non-negative integers we use the prefix-free code $\text{code}_2 = \{1^{|x|}0x \mid x \in \{0,1\}^*\}$. The code of 0 is 100, the code of 1 is 101, and th code of 15 is 111100001.

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

(1) `& R1,R2` is coded in two different ways depending on R2:[4]

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

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

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

$$100.$$

All codings for instruction names and 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.

7. Inductive complexity of the Collatz conjecture

Define the function

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

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.

Next we define the inductive Turing machine $M^{ind,2}_{\text{Collatz}}$ by

$$M^{ind,2}_{\text{Collatz}} = \begin{cases} 0, & \text{if } \forall n \geq 1, C(n) = n, \\ 1, & \text{otherwise}. \end{cases}$$

The inductive program checking the Collatz conjecture based on $M^{ind,2}_{\text{Collatz}}$ is presented in Table 2. The program does not use routines and has 42 instructions and 555 bits, therefore the Collatz conjecture is in the inductive complexity class $\mathfrak{C}^{ind,2}_{U,1}$.

label	instruction	label	instruction	label	instruction
	&OR, 1		&D, 0	L9	+G, F
	&T, 1		&E, 1		+G, F
L1	&OC, 1		&F, 1		+G, 1
	&N, 1	L6	=F, G, L8		=E, E, L3
L2	&G, T		+E, 1	L10	=G, 1, L11
	&K, N		+F, 1		+N, 1
L3	=K, 1, L10		=E, 2, L7		=E, E, L2
	&E, 1		=E, E, L6	L11	&OC, T
L4	&F, E	L7	&E, 0		=OC, T, L12
	+F, 1		+D, 1		&OR, 0
	=F, K, L5		=E, E, L6		=E, E, L13
	+E, 1	L8	=E, 1, L9	L12	+T,1
	=E, E, L4		&G, D		=E, E, L1
L5	&K, E		=E, E, L3	L13	procent

Table 2. Inductive program for the Collatz conjecture

8. Inductive complexity of the twin prime conjecture

The inductive program checking the twin prime conjecture is presented in Table 3.

label	instruction	label	instruction	label	instruction
	=a, a, MAIN		&OE, 1	LH1	&c, LH2
PRIME	&f, 2		&pe, a		&a, ih
LP1	=f, a, LP6	LE1	&c, LE2		=a, a, EFC
	&e, f		&a, pe	LH2	=OE, a, LH3
	&h, 0		=a, a, PRIME		+ih, 1
LP2:	=e, a, LP3	LE2	=d, 1, LE3		=a, a, LH1
	+e, 1	LE5	+pe, 1	LH3	&OH, 1
	+h, 1		=a, a, LE1		&a, ah
	=h, f, LP4	LE3	+pe, 2		&c, ch
	=a, a, LP2		&c, LE4		=a, a, c
LP3	=h, 0, LP5		&a, pe	MAIN	&OM, 1
	+f, 1		=a, a, PRIME		&N, 1
	=a, a, LP1	LE4	=d, 0, LE5	LM1	&a, N
LP4	&h, 0		&OE, ae		&c, LM2
	=a, a, LP2		&a, ae		=a, a, HFC
LP5	&d, 0		&c, ce	LM2	=OH, 1, LM3
	=a, a, LP7		=a, a, c		&OM, 0
LP6	&d, 1	HFC	&ah, a		=a, a, LM4
LP7	=a, a, c		&ch, c	LM3	+N, 1
EFC	&ae, a		&OH, 0		=a, a, LM1
	&ce,c		&ih, 1	LM4	%

Table 3. 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 starts with the instruction labeled MAIN

and has its output register in OM. The program has 63 instructions and 897 bits, therefore is in the inductive complexity class $\mathfrak{C}_{U,1}^{ind,2}$.

9. 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 conjecture and twin prime conjecture. These problems cannot be currently evaluated with the semi-decidability complexity developed in [6, 7, 8, 14, 13], but are both in the lowest inductive complexity class $\mathfrak{C}_{U,1}^{ind,2}$.

There are many open problems. In this paper, as well as in [6, 7, 8, 9], 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?

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, *Internatioanl 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, M. J. Dinneen. A new measure of the difficulty of problems, *Journal for Multiple-Valued Logic and Soft Computing* 12 (2006), 285–307.

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

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

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

[10] C. S. Calude, E. Calude and 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 and C.-K. Shu. Computing a glimpse of randomness, *Experimental Mathematics* 11, 2 (2002), 369–378.

[13] E. Calude. The complexity of Riemann's Hypothesis, *Journal for Multiple-Valued Logic and Soft Computing*, 17, 4 (2011) 1–9.

[14] E. Calude. Fermat's Last Theorem and chaoticity, *Natural Computing*, accepted June 2011.

[15] J. P. Davalan. $3x + 1$, Collatz, Syracuse problem, `http://pagesperso- orange.fr/jean-paul.davalan/liens/liens_syracuse.html`, (accessed on 30 November 2008).

[16] R. K. Guy. Problem E16 in *Unsolved Problems in Number Theory*. Springer, New York, 2004 (third edition), 330–336.

[17] J. Lagarias. The $3x + 1$ problem and its generalizations, *Amer. Math. Monthly* 92 (1985), 3–23.

[18] J. Lagarias (ed.), *The Ultimate Challenge: The* $3x + 1$ *Problem*, AMS, 2010.
[19] 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 Mathematics, University of California, Los Angeles, USA, http://www.math.ucla.edu/~mburgin.

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

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