Parallel Processing Letters
Vol. 23, No. 1 (2013) 1350007 (16 pages)

DOI: 10.1142/S0129626413500072

INDUCTIVE COMPLEXITY OF THE P VERSUS NP PROBLEM*

CRISTIAN S. CALUDE†
Department of Computer Science, University of Auckland, Auckland, and New Zealand
Isaac Newton Institute for Mathematical Sciences, Cambridge, United Kingdom
cristian@cs.auckland.ac.nz

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

and

MELISSA S. QUEEN‡
Department of Computer Science, University of Auckland, Auckland, New Zealand
Dartmouth College, New Hampshire, USA
melissa.s.queen.13@dartmouth.edu

Received October 2012
Revised February 2013
Published 28 March 2013
Communicated by S. Akl

ABSTRACT

This paper does not propose a solution, not even a new possible attack, to the P versus NP problem. We are asking the simpler question: How "complex" is the P versus NP problem? Using the inductive complexity measure—a measure based on computations run by inductive register machines of various orders—developed in [2], we determine an upper bound on the inductive complexity of second order of the P versus NP problem. From this point of view, the P versus NP problem is significantly more complex than the Riemann hypothesis. To date, the P versus NP problem and the Goostein theorem (which is unprovable in Peano Arithmetic) are the most complex mathematical statements (theorems, conjectures and problems) studied in this framework [9, 5, 6, 2, 20].

Keywords: P vs NP problem; inductive complexity; inductive computation.

*An extended abstract of this paper has appeared in J. Durand-Lose, N. Jonoska (eds.). *Proceedings UCNC 2012*, LNCS 7445, Springer, (2012), 2–9.
†Partially supported by a Visiting Fellowship, Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, 2012.
‡Partially supported by a University of Auckland International Summer Scholarship 2012.

1. A class of complexity measures

Mathematics is built upon theorems, conjectures and problems both open and resolved. Some problems intuitively seem highly complex, and have perhaps eluded solution for centuries. Others appear to be less complicated. We would like to be able to quantitatively capture this complexity, and thus be able to compare conjectures from vastly different fields of mathematics. One possible scale we can use has been developed in [9, 5, 6, 2, 8] and applied to different problems in [7, 10, 12, 13, 17]. This method considers the most intuitive way to solve a problem, a brute-force search for a counter-example to the claim. If the conjecture is false, a counter-example will eventually be found. But if a conjecture is true, the search will run on forever. If we could somehow determine ahead of time if the search will run forever, we would be able to prove the conjecture is true. Unfortunately, this equates to solving the halting problem, which is known to be undecidable. But not all is lost, since we are not actually trying to *solve* all mathematical conjectures, but rather to *compare* some of them: indeed, we wish to be able to compare conjectures regardless of their true/false or proven/unproven status.

For this aim we will use a more powerful model of computation than the Turing machine, the inductive computation. The search for a counter-example can be coded into a program, and the program can carefully be encoded into a string of ones and zeroes. Thus for some mathematical conjectures we can create a string of bits (along with an explanation of how to unambiguously read off the program) and say 'if this program halts, the conjecture is false; if it does not halt, the conjecture is true'. It naturally follows that some conjectures can be 'encoded' into bits more simply than others; these conjectures will be considered of low complexity. More complicated conjectures may take a large program and a huge number of bits; these programs are considered to have high complexity. Time complexity plays no role in this analysis.

Although the results we obtain may shed new light on the statements we analyse, they are not intended to solve the problems expressed by those statements, nor to predict how easy/difficult could be to find their solutions.

2. The P versus NP problem

The processing power of computers has grown—and continues to grow—incredibly quickly, and computer-users have become accustomed to newer and faster computers continually being released on the market. In such an environment, it might seem like there is no bound to the size and type of problems that computers can solve—and even if a program runs slowly on today's computers, surely in a few years it will be zipping along on the faster computers of the future. Unfortunately, this is not the case. The problem lies in the asymptotic behaviour of certain algorithms, i.e. their behaviour when the problem instance size gets very large. It makes sense that the larger a problem input size, the longer it takes to solve it, but in some cases the needed time grows faster than we will ever be able to account for with faster computers. The usual solution is to simply find a faster, more efficient algorithm.

But for a large class of problems, many of them of critical practical significance, *no* efficient algorithms have been found. This class is called NP (nondeterministic polynomial), while the class of problems that are known to have efficient algorithms is called P (deterministic polynomial, or, simply, polynomial).

Furthermore, there exists a set of NP problems, called NP-complete, which if one could figure out how to solve just one of them in polynomial-time then we could solve *all* of them in polynomial-time. In asking the question 'Does P=NP?' we are asking if it is possible to solve all NP problems in polynomial-time, or equivalently, if it is possible to solve an NP-complete problem in polynomial-time.

As a concrete example, we present the NP-complete problem used in our program: the subset-sum problem [16] (subsection 35.5: The subset-sum problem). This problem starts with a collection of numbers, and a target number—an instance of the problem—and asks the question: *Does some subset of our collection add up to equal the target?* In small instance sizes this is simple. For example, we can easily check that no subset of (1,2,5) adds up to 4, or that there is a subset of (1,2,5,8) that adds to 7 (namely, 2 and 5). But as the instance size gets larger, the number of possible subsets grows exponentially, and it takes exponential time to check every subset. The brute-force algorithm for solving the subset-sum problem cycles through all subsets of N numbers and, for every one of them, checks if the subset sums to the right number. The running-time is of order $O(N \cdot 2^N)$, since there are 2^N subsets and, to check each subset, we need to sum at most N elements. A faster algorithm proposed by Horowitz and Sahni [21] runs in time $O(2^{N/2})$. If one could show that there is some algorithm that solves every possible instance of subset-sum in polynomial-time, then we would show that P=NP.

The P versus NP problem, formulated independently by Cook [14] and Levin [24], is considered to be one of the most challenging open problems in mathematics. The Clay Mathematics Institute will award a prize of $1.000.000 for its first correct solution, [28]. A constructive proof for P = NP based on an efficient simulation would have significant practical consequences; a proof for P $\neq$ NP (which is widely believed to be the case) would lack practical computational benefits, but would have important theoretical value.

With decades of research dedicated to its resolution, substantial insight was obtained: see more in the official Clay Mathematics Institute presentation of the problem by Cook [15], the papers by Fortnow [18] and Mulmuley [26], Moore-Mertens book [25] (Chapter 6, The Deep Question: P vs NP), and Wöginger's webpage [30].

The use of parallel computers instead of sequential computers does not help because if a problem requires exponential time to be solved on a sequential computer, every parallel computer with finite number of processors solving it runs also in exponential time since the speed-up can be only by a constant factor (e.g. the number of processors). The above result is not "absolute", i.e. it depends on the geometry of the space where the computation is run. More precisely, the above result is true in case the real time-space is Euclidean, and the reason is that the volume

of a sphere of radius r is a polynomial in r, $O(r^3)$. If the real time-space is hyperbolic, the volume of the sphere grows exponentially with r, and this growth can be exploited to dramatically speed-up parallel computations, hence the possibility to solve NP–hard problems in polynomial time. For more details see [23].

Is the polynomial-time algorithm the "correct" mathematical model for feasible computation? An affirmative answer is provided by Cobham's thesis which states that "P" means *easy* while "the complement of P" means *hard*. Is Cobham's thesis as "credible" as the Church-Turing thesis, which deals with computability in principle, i.e. by disregarding resources? According to Davis [22] (p. 568–569) the answer is negative[a]. In fact, we believe that the concept of feasible computation like the concept of randomness are paradoxical (blind spots in the terminology of [3]): they cannot be grasped in finite words. More precisely, *we conjecture that there is no good mathematical model for feasible computation*. From this perspective, the P versus NP problem is less a computer science problem than a mathematical one.

3. Goal

By measuring the complexity of the P versus NP problem we hope to shine a little more light on the problem; certainly, this is not an attempt to solve it. To do this, we have developed an inductive register machine program that searches for a counter-example to the claim that "P $\neq$ NP". This counter-example would be a program that runs in polynomial-time for all instances of the subset-sum problem, our choice of NP-complete problem. The register machine program has a prefix-free binary encoding, and the length of this string determines an upper bound of the complexity class of the P versus NP problem.

4. Method

The register machine language we use is a refinement, constructed in [6], of the language in [9]; see also [17]. The register machine language is simple and minimal (each instruction is essential; no instruction can be reduced to a combination of the other instructions). It consists of the following instructions:

= R1,R2,R3 If the content of R1 and R2 are equal, then the execution continues at the R3rd instruction of the program. If the contents of R1 and R2 are not equal, then execution continues with the next instruction in sequence.

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

[a]In the discussions following J. Hartmanis' invited lecture *Turing Machine Inspired Computer Science Results*, CiE2012, 22 June 2012, `http://www.mathcomp.leeds.ac.uk/turing2012/WScie12/Content/abstracts/juris.html`, M. Davis asked the question he posed the speaker about 30 years ago: "How would you feel if P=NP with a polynomial of degree 100?" Hartmanis' original answer was: "God cannot be so cruel!".

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

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

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

A register machine program is a finite list of these instructions. It is allowed access to an arbitrary number of registers, and each register can hold an arbitrarily large positive integer. The prefix free binary encoding of these instructions is discussed in detail in [5, 6], and briefly below.

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

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

Programs often need to execute the same operations many different times, and it is convenient to create routines for these operations. Routines that our program uses include MUL (multiply), POW (power/exponentiation), CMP (compare), SUBT (subtraction) and DIV2 (halves a number).

As a concrete example, the subtraction routine is given in Table 1. The routine SUBT uses registers `a` through `e`, computes `a` − `b`, stores the answer in `d`, then returns to the line number stored in `c`. It assumes that `a` ≥ `b`.

The register names, `a` = 010, `b` = 00100, `c` = 00101, `d` = 00111, and `e` = 011, were chosen to minimise the overall number of bits used. In total, this routine is represented by the 70 bit string:

0100111100010110011111000110010010101101000101100001111011010100101101 0

Table 1. SUBT

Label	Instruction	Comments	Binary representation
SUBT1	`& d, 0`		`01 00111 100`
LS1	`& e, d`		`01 011 00111`
	`+ e, b`		`100 011 00100`
	`= e, a, c`	`// d+b=a`	`101 011 010 00101`
	`+ d, 1`		`100 00111 101`
	`= a, a, LS1`	`// loop`	`101 010 010 11010`

5. Register machine language implementation of arrays

We use the coding for the array data structure library developed by Dinneen [17] which represents arrays (lists) in a single register variable. An integer element a_i within an array A is represented as a sequence of a_i bits 0; the bit 1 is used as a (leading) separator or deliminator of the array elements. If there are no 1's (e.g. the register has value 0) then we have an array of size 0. For example, the array [1,4,0] is encoded 10100001 or 11000010, depending on chosen endianness (the encoding is from left to right or vice-versa).

A number of array operations are used frequently, and these have been packaged into subroutines. In our particular program we make use of SIZE (returns the size of the array), APPEND (appends one element), ELM (returns the element at a particular index) and RPL (replaces the element at a particular index).

6. From standard computation to inductive computation

The main program for the P versus NP problem consists of two nested loops. The outer loop tests every program-polynomial tuple. For each program and polynomial, the inner loop checks if the program can solve all instances of the subset-sum problem in polynomial steps or less. In the usual model of computation these nested loops have a serious pitfall: The program may run forever for two different reasons. It may run forever because it never finds a program that works (there are infinitely many programs), in which case P does not equal NP. The second reason it may run forever is because it *has* found a program that works, and since there are an infinite number of instances of the subset-sum problem, it loops forever testing all of them.

To resolve this issue, we chose to use a slightly modified version of computation: the *inductive computation* [1]. Under this model, a program is allowed to run forever but still be considered to give an answer if, after a finite number of steps, the *output stabilises*. To make our program suitable for an inductive register machine program, we must modify each loop in the following way: If the loop is successfully running, write a 1 into the output register, otherwise when the loop halts write a 0 into the output register (and stop looping). We thus ensure that the output register will not oscillate, and under the inductive computation model it will always return a result. Namely, the output will be 1 if the loop runs forever, and 0 if at some point it will halt.

Finite standard Turing and inductive computations produce the same results;

however, the inductive computation is more powerful than standard computation as in the former case some infinite computations can produce outputs.

In what follows we will use the above register machine language as a *universal prefix-free inductive machine* U^{ind} (see more in [2]). This type of computation gives rise to an *inductive complexity measure*.

7. An inductive register program for P versus NP

It is easy to note that P $\neq$ NP if and only if every polynomial-time program (i.e. a pair consisting of a program and a polynomial controlling the time of the execution of the program) cannot solve at least one instance of the subset-sum problem. Hence the P versus NP problem can be represented by a Π_2–sentence, i.e. a sentence of the form $\forall n \exists i R(n,i)$, where R is a computable predicate.[b] Starting from a representation $\forall n \exists i R(n,i)$ (where R is a computable predicate) of the P versus NP problem, we construct the inductive register machine program 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 register 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,

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

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

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

MAIN, the main algorithm for $M_R^{ind,2}$ that solves the P versus NP problem, is presented in the algorithm below. As we have already mentioned, the program consists of two nested loops; the outer loop goes through all possible program and polynomial pairs, and the inner loop runs the program with every possible instance of the subset-sum problem, letting it execute at most polynomial steps for each instance.

Our algorithm requires the testing of the correctness of every possible polynomial-time program for each instance of the subset-sum problem. Is it possible

[b]By now the reader has understood that the word "problem" has different meanings in Cook's theory and in our complexity analysis.

MAIN: RESULT IS 1 IF P≠NP, 0 IF P=NP

```
// Z is the output register, while the loop is running it is set to 1
Z ← 1
for all tuples (C,J,P) do
  // Now we run the simulation (also on an inductive Turing machine)
  run SIM
  // check the result register (Y)
  if Y = 1 then
    // found a polynomial-time algorithm, P=NP
    Z ← 0
    HALT
  else
    // that program didn't work, try the next one
    continue
  end if
end for
```

SIM: RESULT IS 1 IF PROGRAM P SUCCEEDS IN POLYNOMIAL-TIME, 0 IF NOT

```
// Y is the output register, while the loop is running it is set to 1
Y ← 1
for all instances S of subset-sum do
  Simulate program P with input S for at most (C ∗ (|S|^J + 1)) steps.
  if P executed without error and calculated the correct answer then
    continue to next instance
  else
    // This program doesn't work, stop looping
    Y ← 0
    return
  end if
end for
```

to achieve this testing given that the correctness problem is undecidable? The answer is affirmative because we are dealing with time-controlled computations, each running in a finite amount of time; the correctness test may take in some cases exponential time to complete.

It is important to note that the correctness of the polynomial-time program is established when it runs accurately on *all possible instances* of subset-sum problem. It is not enough for the program to run correctly in only some cases. Indeed, since we loop through all possibly instances, we will eventually come across the cases in which an invalid program fails. In particular, programs that randomly "guess", or that always give the same answer eventually fail.

The program-polynomial tuples are generated by incrementing through the natural numbers, treating each number as an array and asking if that array has three elements. Non-complying numbers are ignored; otherwise, we consider the first and second elements to be `C` and `J` respectively, which define the polynomial[c] $\texttt{C}*(x^{\texttt{J}}+1)$, and the third element to be the program `P`. To enumerate all instances of subset-sum problem, we similarly go through the natural numbers and interpret them as arrays with at least 2 elements. For each array we ask the question: Does some subset of its first $(N-1)$ elements sum to the N^{th} element, where N is the size of the array?

When simulating the program `P` we give it access to an unlimited number of registers which are stored in an array `R`. The unique coding of a register name is used to represent the index of that register in the array `R`, and the array dynamically grows to include all possible simulated registers. After running the program `P` we assume that its answer to the subset sum instance is in the register encoded as 010, which corresponds to `R`[2][d]. One can easily check if the proposed answer is correct.

As an example, consider the 672^{nd} loop of the outer (main) program. In binary, this corresponds to the number 101010, which, interpreted as an array is $[1, 1, 1]$, meaning $\texttt{C} = 1$, $\texttt{J} = 1$ and $\texttt{P} = 1$. In binary, the program is simply 1.

The outer program then executes the simulation on another inductive Turing machine. The first instance of subset-sum that it will test is 11, the two element array $[0, 0]$. It asks: Is there a subset of elements in [0] that add up to 0? Clearly, the answer is positive. In preparation for simulation, the input for program `P` is made prefix free: 11 becomes 11011, so the program is expected to read five bits (which are read right to left) before halting. The simulation begins, but the parser quickly realises that 1 is not a valid program, and will branch to an error statement. In this statement, the inductive Turing machines writes 0 to its output register (named `Y` in the pseudocode) and halts. The main loop (the outer inductive Turing machine) queries the output register, reads the zero and continues on to the next loop.

In the simulation process every syntactical error is detected. The simulated program may fail because of compile or run time errors, e.g. invalid use of a register/literal encoding, the halt instruction appears in the middle of a program, branching out of bounds, not reading all input bits. The program is also disqualified if, after the prescribed polynomial number of steps, it has not naturally halted.

8. An upper bound for the inductive complexity of P versus NP

To every mathematical sentence of the form $\rho = \forall n \exists i R(n, i)$, where $R(n, i)$ is a computable predicate—called Π_2–sentence, we associate the inductive register machine of second order $M_R^{ind,2}$ as above.

[c]Obviously, in this way we cover all possible run-time polynomials.

[d]There is no register named 00 or 01, so `R`[2] is the first possible register that the simulated program could use. Of course, it may not ever use this register, in which case `R`[2] would always be 0, the default initial value.

Note that there are many programs for the universal prefix-free inductive machine U^{ind} which implement $M_R^{ind,2}$. For each of them we have:

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

The inductive complexity measure of second order is defined by:

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

and, correspondingly, the *inductive complexity class of second order* is:[e]

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

The complexity measure, as stated, is unfortunately *incomputable* (see [4]), so we resort to measuring upper bounds of the complexity. This is still a useful measurement and allows us to rank and compare Π_2–sentences [5].

The inductive register program for the P versus NP problem consists of 362 instructions. More details about the program, including flowcharts for the main program, principal routines, simulation, run and commands for the instructions &, +, =, !, as well as a fully commented version of the program are presented in Appendices A and B in [11]. The syntactical correctness of the program was checked using a tool developed in [19].

```
1     = a a 265      18    = b e c      35    + a 1
2     & d 1          19    + e          36    = a a 28
3     & e 0          20    & d 0        37    & as a
4     = e b c        21    & e d        38    & bs b
5     + e 1          22    + e b        39    & cs 0
6     & f 1          23    = e a c      40    & b 42
7     & dp d         24    + d 1        41     = a a 26
8     = f a 4        25    = a a 21     42     + cs c
9     + d dp         26    & ad a       43     = a 0 45
10    + f 1          27    & a 0        44     = a a 21
11    = a a 8        28    & c 0        45     & a as
12    & d 0          29    & e a        46     & b bs
13    = a b c        30    + e e        47     & c cs
14    & e 0          31    = e ad b     48     = a a b
15    & d 1          32    & c 1        49     & ap 0
16    = a e c        33    + e 1        50     + a a
17    & d 2          34    = e ad b     51     + a 1
```

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

```
52    = ap b 56
53    + a a
54    + ap 1
55    = a a 52
56    = a a c
57    & ae a
58    & be b
59    & ce c
60    & b 62
61    = a a 37
62    & a c
63    & b I
64    & c 66
65    = a a 20
66    & ie d
67    & d 0
68    = ie d 73
69    & b 71
70    = a a 26
71    + d c
72    = a a 68
73    & d 0
74    & b 76
75    = a a 26
76    = c 1 79
77    + d 1
78    = a a 74
79    & a ae
80    & b be
81    = a a ce
82    & ar a
83    & br b
84    & cr c
85    & ir I
86    & b 88
87    = a a 37
88    & nr c
89    & I 0
90    & fr 0
91    = I nr 107
92    = I ir 103
93    & a ar
94    & c 96
95    = a a 57
96    & b d
97    & a fr
98    & c 100
99    = a a 49
100   & fr a
101   + I 1
102   = a a 91
103   & b br
104   & a fr
105   & c 100
106   = a a 49
107   & a fr
108   & b br
109   & c cr
110   & I ir
111    = a a c
112   & bg b
113   & ISLIT 0
114   & b 116
115   = a a 26
116   & b bg
117   + a a
118   = c 0 138
119   & ISLIT 1
120   + a 1
121   & brl b
122   & LIT 0
123   = a 0 321
124   & irl 0
125   & b 127
126   = a a 26
127   + irl 1
128   = c 0 130
129   = a a 125
130   & jrl 0
131   + jrl 1
132   + LIT LIT
133   = c 0 135
134   + LIT 1
135   = irl jrl brl
136   & b 131
137   = a a 26
138   & brr b
139   & REG 0
140   = a 0 321
141   & irr 0
142   & b 144
143   = a a 26
144   = c 1 321
145   + irr 1
146   = c 1 149
147   & b 145
148   = a a 26
149   & jrr 0
150   + jrr 1
151   + REG REG
152   = c 0 154
153   + REG 1
154   = irr jrr 157
155   & b 150
156   = a a 26
157   & arr a
158   & a R
159   & b 160
160   = a a 37
161   & a c
162   & b REG
163   & c 165
164   = a a 12
165   = d 1 168
166   & a arr
167   = a a brr
168   + R R
169   + R 1
170   = a a 158
171   & bp b
172   = T XT 321
173   & a P
174   & ip 1
175   = a 0 240
176   = ip NEXT 179
177   & e 0
178   = a a 181
179   & e 1
180   + T 1
```

```
181    + ip 1
182    & b 52
183    = a a 26
184    = c 0 56
185    & b 187
186    = a a 26
187    = c 0 321
188    & b 190
189    = a a 26
190    = c 0 204
191    = e 1 196
192    & b 194
193    = a a 138
194    & b 175
195    = a a 112
196    & rn 198
197    = a a 242
198    + LIT2 LIT1
199    & I REG1
200    & a R
201    & b LIT2
202    & c 171
203    = a a 82
204    = e 1 207
205    & b 175
206    = a a 138
207    + NEXT 1
208    & b 210
209    = a a 138
210    = IN 0 321
211    & a IN
212    & b 214
213    = a a 26
214    & b c
215    & a R
216    & I REG
217    & c 171
218    = a a 82
219    & b 221
220    = a a 26
221    = c 0 226
222    = e 1 224
223    = a a 191
224    & rn 199
225    = a a 242
226    = e 1 232
227    & b 229
228    = a a 138
229    & b 231
230    = a a 112
231    = a a 194
232    & rn 234
233    = a a 242
234    = LIT1 LIT2 236
235    = a a 171
236    & rn 238
237    = a a 253
238    & NEXT LIT2
239    = a a 171
240    = ip NEXT bp
241    = a a 321
242    + NEXT 1
243    & b 245
244    = a a 138
245    & REG1 REG
246    & I REG1
247    & ax a
248    & a R
249    & c 251
250    = a a 57
251    & LIT1 d
252    & a ax
253    & b 255
254    = a a 112
255    & ax a
256    = ISLIT 1 262
257    & I REG
258    & a R
259    & c 261
260    = a a 57
261    & LIT d
262    & LIT2 LIT
263    & a ax
264    = a a rn
265    & PP 0
266    & Z 0
267    + PP 1
268    & a PP
269    & b 271
270    = a a 37
271    = c 3 273
272    = a a 267
273    & I 1
274    & c 276
275    = a a 57
276    & C d
277    & I 2
278    & c 280
279    = a a 57
280    & J d
281    & I 3
282    & c 284
283    = a a 57
284    & P d
285    = a a 289
286    = Y 0 267
287    & Z 1
288    = a a 362
289    & S 0
290    & Y 1
291    + S 1
292    & a S
293    & b 295
294    = a a 37
295    & N c
296    = N 1 291
297    & a N
298    & b J
299    & c 301
300    = a a 2
301    + d 1
302    & e 0
303    & XT 0
304    = e C 308
305    + XT d
306    + e 1
307    = a a 304
308    & IN S
309    + IN IN
```

```
310    & i 0          328    = a a 2
311    = i N 316      329    & da d
312    + i 1          330    & e 0
313    + IN IN        331    & f 0
314    + IN 1         332    + e 1
315    = a a 311      333    = e da 356
316    & T 0          334    & a e
317    & R 3          335    & I 0
318    & NEXT 1       336    & s 0
319    & b 323        337    = a 0 332
320    = a a 171      338    + I 1
321    & Y            339    & b 341
322    = a a 286      340    = a a 26
323    = IN 0 325     341    = c 0 337
324    = a a 321      342    & aa a
325    & a 2          343    & a S
326    & b N          344    & c 346
327    & c 329        345     = a a 57
```

```
346     + s d
347     & a aa
348     = a a 337
349     & a S
350     & I N
351     & c 353
352     = a a 57
353     = s d 355
354     = a a 332
355     & f 1
356     & I 2
357     & a R
358     & c 291
359    = a a 57
360    = d f 291
361    = a a 321
362    %
```

The first ten instructions in the program for the P versus NP problem are presented in machine code followed by the length of the corresponding binary encoding (accruing to the scheme presented in Section 4.).

```
Instruction number=1
= 00
a 010
a 010
265 11111111000001011
The number of bits 25

Instruction number=2
& 01
d 00110
1 101
The number of bits 10

Instruction number=3
& 01
e 00101
0 100
The number of bits 10
```

```
Instruction number=4
= 00
e 00101
b 011
c 00100
The number of bits 15

Instruction number=5
+ 111
e 00101
1 101
The number of bits 11

Instruction number=6
& 01
f 0001011
1 101
The number of bits 12
```

```
Instruction number=7
& 01
dp 00000101100
d 00110
The number of bits 18

Instruction number=8
= 00
f 0001011
a 010
4 11010
The number of bits 17

Instruction number=9
+ 111
d 00110
dp 00000101100
The number of bits 19

Instruction number=10
+ 111
f 0001011
1 101
The number of bits 13
```


which leads to the first 150 bits of the code for the P versus NP program:

000100101111111100000101101001101010100101100000010101100100111001011010100010111010100000101100001100000010110101101011100110000001011001110001011101

After optimising the coding according to the frequency of registers, the code of the inductive register machine program consists of 6,495 bits,[f] putting the P versus NP problem into the *inductive complexity class of second order 7*. The full binary encoding of the program is in Appendix E of [11].

The Riemann hypothesis, another problem on the list of the Clay Mathematics Institute millennium open problems [29] and arguably the most important open problem in mathematics, is in the *inductive complexity class of first order 3*, a significantly lower complexity class. Goodstein theorem—which is unprovable in Peano Arithmetic—is also in *the inductive complexity class of second order 7*. The Collatz conjecture and the twin-prime conjecture fall into the first inductive complexity class of second order, cf.[2]; all other problems studied till now fall into inductive complexity classes of first order smaller than 4, cf. [7, 10, 12, 13, 17].

There are probably further modifications that can be made to shorten the program, possibly by improving the simulation potential polynomial-time programs and/or by using a different NP–complete problem. Our analysis has used the representation of the P versus NP problem as a Π_2–sentence; it is an open question whether *the P versus NP problem is a* Π_1*–sentence*, i.e. of the form $\forall n P(n)$, where P is a unary predicated.

We reiterate that with the inductive complexity we compare problems from one point of view only, and the upper-bounds obtained, reflecting the current knowledge of those problems, are time-dependent. In this context it would be interesting to

[f]This code is with 296 bits shorter than the one announced in the *UCNC 2012* extended abstract.

find lower bounds of the inductive complexity for the mathematical problems for which upper bounds have been estimated.

Acknowledgments

We thank M. Dinneen, J. Hertel and, particularly, S. Rudeanu, for many comments and suggestions which improved the paper. A. Gandhi and H. Naderi have developed the tool described in [19]. We also thank the anonymous referees for their useful comments.

References

[1] M. Burgin. *Super-recursive Algorithms*, Springer, Heidelberg, 2005.
[2] M. Burgin, C.S. Calude, E. Calude. Inductive complexity measures for mathematical problems, *International Journal of Foundations of Computer Science*, accepted.
[3] W. Byers. *The Blind Spot: Science and the Crisis of Uncertainty*, Princeton University Press, Princeton, 2011.
[4] C. S. Calude. *Information and Randomness: An Algorithmic Perspective,* 2nd Edition, Revised and Extended, Springer-Verlag, Berlin, 2002
[5] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 1 *Complex Systems*, 18-3 (2009), 267–285.
[6] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 2 *Complex Systems*, 18-4, (2010), 387–401.
[7] C. S. Calude, E. Calude. The complexity of the Four Colour Theorem, *LMS J. Comput. Math.* 13 (2010), 414–425.
[8] C. S. Calude, E. Calude. Algorithmic complexity of mathematical problems: an overview of results and open problems, *International Journal of Unconventional Computing*, accepted.
[9] C. S. Calude, E. Calude, M. J. Dinneen. A new measure of the difficulty of problems, *Journal for Multiple-Valued Logic and Soft Computing* 12 (2006), 285–307.
[10] C. S. Calude, E. Calude, M. S. Queen. The complexity of Euler's integer partition theorem *Theoretical Computer Science*, 2012, doi: 10.1016./j.tcs.2012.03.02.
[11] C. S. Calude, E. Calude, M. S. Queen. The Complexity of the P vs NP Problem, *CDMTCS Research Report* 429, 2012, 39 pp.
[12] C. S. Calude, E. Calude, K. Svozil. The complexity of proving chaoticity and the Church-Turing Thesis, *Chaos* 20 037103 (2010), 1–5.
[13] E. Calude. The complexity of Riemann's Hypothesis, *Journal for Multiple-Valued Logic and Soft Computing*, 18 (3-4) (2012), 257–265.
[14] S. Cook. The complexity of theorem proving procedures, in *STOC '71, Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ACM New York, NY, USA, 1971, 151–158.
[15] S. Cook. The P versus NP Problem, manuscript, 12 pages, `http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf`, visited on 16 June 2012.
[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* MIT Press and McGraw-Hill, 2011. 2nd ed.
[17] 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.
[18] L. Fortnow. The status of the P vs NP problem, *CACM* 52, 9 (2009), 78–86.

[19] A. Gandhi. *Register Machine Syntax Checker and Translator: Manual*, University of Auckland, Version 0.0.3.0, April, 2012.
[20] J. Hertel. Inductive complexity of Goodstein's theorem, in J. Durand-Lose, N. Jonoska (eds.). *Proceedings UCNC 2012*, LNCS 7445, Springer, 2012, 141–151.
[21] E. Horowitz, S. Sahni. Computing partitions with applications to the knapsack problem, *JACM* 21(1974), 277–292.
[22] A. Jackson. Interview with Martin Davis, *Notices AMS* 55, 5 (2008), 560–571.
[23] V. Kreinovich, M. Margenstern. In some curved spaces, one can solve NP-hard problems in polynomial time, *Journal of Mathematical Sciences* 158, 5 (2009), 727–740. (Originally published in Russian in *Zapinski Nauchnykh Seminarov POMI*, 358 (2008), 224-250.)
[24] L. Levin. Universal search problems, *Problemy Peredachi Informatsii* 9 (1973), 265–266 (in Russian). English translation in [27].
[25] C. Moore, S. Mertens. *The Nature of Computation*, Oxford University Press, Oxford, 2011.
[26] K. D. Mulmuley. The GCT program toward the P vs NP problem, *CACM* 55, 6 (2012), 98–107.
[27] B. A. Trakhtenbrot. A survey of Russian approaches to Perebor (brute-force search) algorithms, *Annals of the History of Computing* 6 (1984), 384–400.
[28] `http://www.claymath.org/millennium/P_vs_NP/`, visited on 16 June 2012.
[29] `http://www.claymath.org/millennium/Riemann_Hypothesis/`, visited on 16 June 2012.
[30] G. J. Wöginger. The P-versus-NP webpage, `http://www.win.tue.nl/~gwoegi/P-versus-NP.htm`, visited on 16 June 2012.