Inductive Complexity of P versus NP Problem
Extended Abstract

Cristian S. Calude[1,⋆], Elena Calude[2], and Melissa S. Queen[3,⋆⋆]

[1] Department of Computer Science, University of Auckland, Auckland, New Zealand
Isaac Newton Institute for Mathematical Sciences, Cambridge, United Kingdom
cristian@cs.auckland.ac.nz

[2] Institute of Information and Mathematical Sciences,
Massey University at Auckland, New Zealand
e.calude@massey.ac.nz

[3] Department of Computer Science, University of Auckland, Auckland, New Zealand
Dartmouth College, New Hampshire, USA
melissa.s.queen.13@dartmouth.edu

Abstract. Using the complexity measure developed in [7,3,4] and the extensions obtained by using inductive register machines of various orders in [1,2], 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 more complex than the Riemann hypothesis.

1 A Complexity Measure

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 [7,3,4,2,6] and applied to different problems in [5,8,9,11,15]. 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 counterexample 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.

⋆ 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.

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 any mathematical conjecture, 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.

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 [14] (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 $2N$ subsets and, to check each subset, we need to sum at most N elements. A faster algorithm proposed by Horowitz and Sahni [17] 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 [12] and Levin [19], 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, [23]. 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 [13], the papers by Fortnow [16] and Mulmuley [21], Moore-Mertens book [20] (Chapter 6, The Deep Question: P vs NP), and Wöginger's webpage [25].

Is the polynomial-time algorithm the "correct" mathematical model for feasible computation? Is this formalisation as "credible" as the Church-Turing thesis, which deals with computability in principle, i.e. by disregarding resources? According to Davis [18] (p. 568–569) the answer is negative[1]; 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 nature of the problem. To do this, we have developed an inductive register machine program that searches for a counter-example to the claim that "P does not equal 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 [4], of the language in [7]; see also [15]. It consists of the following instructions:

[1] 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,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.

+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 [3,4].

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), DIV2 (halves a number), as well as routines used to interact with arrays: ELM (get an element), SIZE (size of the array), APPEND (add an element), and RPL (replace an element).

5 From Standard Computation to Inductive Computation

Our main program consists of two nested loops. The outer loop tests every program-polynomial tuple. For each program and polynomial, the inner loop checks to see 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. But, it may run forever because is *has* found a program that works, but 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 it will at some point halt.

Needless to say, inductive computation is more powerful than standard Turing computation.

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.*

6 An Inductive Register Program for P versus NP

The P versus NP problem is a Π_2–sentence, i.e. a sentence of the form $\forall n \exists i R(n,i)$, where R is a computable predicate. From this representation 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,

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

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 $T_R^{ind,1}$.

MAIN, the main algorithm for $M_R^{ind,2}$ that encodes 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. 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 of the cases, and 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.

MAIN: RESULT IS 1 IF P≠NP, 0 IF P=NP

```
// Z is our 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 our 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
```

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[2] $\mathtt{C} * (x^{\mathtt{J}} + 1)$, 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 pose 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`. 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]. One can easily check if the proposed answer is correct.

7 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, we associate the inductive register machine of second order $M_R^{ind,2}$ as above. Note that there are many programs for 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:

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

The complexity measure, as stated, is unfortunately *incomputable* (see [10]), so we resort to measuring upper bounds of the complexity. This is still a useful measurement and allows us to rank and compare conjectures [3].

The inductive register program based on the main algorithm described above consists of 368 instructions and was encoded with 6791 bits, putting the P versus NP problem into the *inductive complexity class of second order 7*. The Riemann hypothesis, another problem on the list of the Clay Mathematics Institute millennium open problems [24] and arguably the most important open problem mathematics, is in the *inductive complexity class of first order 3*.

As with all complexity measures of this type, this is only an upper bound. There are very 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.

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

References

1. Burgin, M.: Super-recursive Algorithms. Springer, Heidelberg (2005)
2. Burgin, M., Calude, C.S., Calude, E.: Inductive Complexity Measures for Mathematical Problems. CDMTCS Research Report 416, 11 (2011)
3. Calude, C.S., Calude, E.: Evaluating the complexity of mathematical problems. Part 1. Complex Systems 18(3), 267–285 (2009)
4. Calude, C.S., Calude, E.: Evaluating the complexity of mathematical problems. Part 2. Complex Systems 18(4), 387–401 (2010)
5. Calude, C.S., Calude, E.: The complexity of the Four Colour Theorem. LMS J. Comput. Math. 13, 414–425 (2010)
6. Calude, C.S., Calude, E.: The Complexity of Mathematical Problems: An Overview of Results and Open Problems. CDMTCS Research Report 410, 12 (2011)
7. Calude, C.S., Calude, E., Dinneen, M.J.: A new measure of the difficulty of problems. Journal for Multiple-Valued Logic and Soft Computing 12, 285–307 (2006)
8. Calude, C.S., Calude, E., Queen, M.S.: The complexity of Euler's integer partition theorem. Theoretical Computer Science (2012), doi:10.1016./j.tcs.2012.03.02
9. Calude, C.S., Calude, E., Svozil, K.: The complexity of proving chaoticity and the Church-Turing Thesis. Chaos 20, 037103, 1–5 (2010)
10. Calude, C.S.: Information and Randomness: An Algorithmic Perspective, 2nd edn. Springer, Berlin (2002) (revised and extended)
11. Calude, E.: The complexity of Riemann's Hypothesis. Journal for Multiple-Valued Logic and Soft Computing 18(3-4), 257–265 (2012)
12. Cook, S.: The complexity of theorem proving procedures. In: STOC 1971, Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 151–158. ACM, New York (1971)
13. Cook, S.: The P versus NP Problem, 12 pages (manuscript), `http://www.claymath.org/millennium/P_vs_NP/pvsnp.pdf` (visited on June 16, 2012)
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill (2001) [1990]
15. Dinneen, M.J.: A Program-Size Complexity Measure for Mathematical Problems and Conjectures. In: Dinneen, M.J., Khoussainov, B., Nies, A. (eds.) Computation, Physics and Beyond. LNCS, vol. 7160, pp. 81–93. Springer, Heidelberg (2012)
16. Fortnow, L.: The status of the P vs NP problem. CACM 52(9), 78–86 (2009)
17. Horowitz, E., Sahni, S.: Computing partitions with applications to the knapsack problem. JACM 21, 277–292 (1974)
18. Jackson, A.: Interview with Martin Davis. Notices AMS 55(5), 560–571 (2008)
19. Levin, L.: Universal search problems. Problemy Peredachi Informatsii 9, 265–266 (1973) (in Russian), English translation in [22]
20. Moore, C., Mertens, S.: The Nature of Computation. Oxford University Press, Oxford (2011)
21. Mulmuley, K.D.: The GCT program toward the P vs NP problem. CACM 55(6), 98–107 (2012)
22. Trakhtenbrot, B.A.: A survey of Russian approaches to Perebor (brute-force search) algorithms. Annals of the History of Computing 6, 384–400 (1984)
23. `http://www.claymath.org/millennium/P_vs_NP/` (visited on June 16, 2012)
24. `http://www.claymath.org/millennium/Riemann_Hypothesis/` (visited on June 16, 2012)
25. Wöginger, G.J.: The P-versus-NP webpage, `http://www.win.tue.nl/~gwoegi/P-versus-NP.htm` (visited on June 16, 2012)