My new constructive Kannan's theorem construction using more standard conventions

Sunny Daniels
e-mail: sdaniels@lycos.com

Abstract:

I appreciate the feedback from Fortnow (the author of [1]) on my recent viXra article [2], and in particular him supplying me with a definition of the standard polynomial-time universal Turing machine. In this article I re-work my previous definitions and proofs to put in the circuit size exponent "k" explicitly (rather than the pseudo-variable "99" in [2]) and use more standard conventions for the derivation of my construction from the standard polynomial-time universal Turing machine construction.

Introduction:

I appreciate the feedback from Fortnow (the author of [1]) on my recent viXra article [2], and in particular him supplying me with a definition of the standard polynomial-time universal Turing machine. In this article I rework my definitions and proofs from [2] to make them (hopefully) easier to understand and more in line with the standard construction of the universal polynomial-time Turing machine supplied by Fortnow in his recent e-mail correspondence with me.

The Standard Polynomial-Time Universal Turing Machine

Fortnow supplied me with the URL [3], in my recent e-mail correspondence with him, for a "standard" universal-time NP Turing machine. He describes his construction this way:

$L = \{(<M>,x,1^k) \mid M \text{ is a nondeterministic machine and } M(x) \text{ accepts in } k \text{ steps}\}$

Here 1^k is just a string consisting of exactly k 1s.

In order to avoid potential confusion with our later use of "k" as the exponent for a circuit size bound, we change "k" to "q" here:

$L = \{(<M>,x,1^q) \mid M \text{ is a nondeterministic machine and } M(x) \text{ accepts in } q \text{ steps}\}$

Here 1^q is just a string consisting of exactly q 1s.

Here we are simulating a \Sigma_2 machine rather than an NP machine, so we change the construction to (sorry, I am being sloppy here and using pseudo-LaTeX names like "\Sigma_2" rather than putting in superscripts and subscripts properly; I am writing this with LibreOffice under Linux. I am happy to fix this if readers want):

L_2 = {(<M>,x,1^q) | M is a \Sigma_2 machine and M(x) accepts in q steps}

Here 1^q is just a string consisting of exactly q 1s.

(Again about my sloppy use of L_2 for L subscript 2 rather than doing it properly with the equation editor).

Here we assume that <M>, the encoding of the control unit of the machine M, is just a string of 1s and 0s.

Here we are looking at simulating machines like the machine that accepts L_2 with circuits, so we need to fix an encoding of inputs to L_2 as strings of 1s and 0s. We have five symbols in L_2's tape alphabet (if that is the right word for it), so we need at least three bits per symbol in L_2's tape alphabet (2^2 is 4, 2^3 is 8). We call this encoding function standard_encoding_of. An example of a defintion that, if I am not mistaken, would work is:

```
0 by 000
1 by 001
( by 010
, by 011
) by 100
101 unused
110 unused
111 unused
```

We will call this point in the definition (***STANDARD ENCODING DEFINITION***). We will refer back to it later. If any of our machines using standard_encoding_of find 101, 110 or 111 (the unused encodings) in their input, they halt and reject immedately.

The Construction Itself:

I agreed with Fortnow, in my recent e-mail discussion, that the extension of this to my construction (putting the running time bound input through a polynomial before using it as the running time bound) would be:

L_3 = {(<M>,x,1^q) | M is a \Sigma^2 machine and M(x) accepts in P(q) steps}

Where P is the polynomial discussed in this context in my earlier viXra article, which we will define again below.

To make this even clearer, the construction in my earlier ViXra article re-worked for L_3 would be:

1) The Σ_2^P machine in question is a universal alternating Turing machine that is both time-bounded and alternation-bounded, in a particular way.

2) An example of an input that it could maybe possibly accept is

the encoding of, using standard_encoding_of defined above:

(0010110100011101101,001011101010101,111111)

where:

0010110100011101101 is the machine control unit (would probably have to be a lot longer than this in practice to be able to compute anything useful)

001011101010101 is the input to the machine.

111111 is the time bound function (P) input in unary notation (i.e. 6). Again it would probably have to be a lot bigger than 6 in practice to be able to compute anything useful.

3) In general, the machine uses the set of tape symbols {blank, 0, 1} and will certainly reject unless the input is of the form: standard_encoding_of((<M>,x,1^q))

Where (sorry, this time out of order):

2a) 1^q is a string of 1s for the input to the time bound polynomial which we will discuss later.

2b) <M> is representation of control unit of tm being simulated, as 1s and 0s.

x is input to machine being stimulated.

3) The alternation count of the machine being simulated is capped at 1: existential first then universal. This alternation count is kept in a variable somewhere on the tape of the simulating machine. If the simulated machine tries to enter a second existential state then the simulating machine will halt and reject. This behaviour of the simulating machine, together with the polynomial time bound condition below, if I am not mistaken, ensures that the simulating machine is in Σ_2^P .

3) A polynomial time bound P of the simulation (as function of q) is hard coded into simulating machine, which accepts the language L_3.

Proof that the Construction Works:

So let any positive integer k be given.

Suppose that our machine L_3 can be simulated by a CCT family C of size O(n^k).

Then we connect inputs of the appropriate circuit in C as follows,

for any given Σ_2 machine N (that is any alternating Turing machine N that starts off in the existential state and then does at most one alternation, that alternation if it happens taking N into the universal state):

a) The first three inputs of the circuit in C are hardwired to standard_encoding_of(() (standard_encoding_of the (symbol, that is), i.e. 010 (look back at ***STANDARD_ENCODING_DEFINITION***).

b) Let r be the length (in 0s and 1s BEFORE putting through the function standard_encoding_of) of <N>, i.e. a representation, according to the convention that we used in the definition of L_3 above, of the control unit of N. Then we hardwire the following 3r inputs to the circuit in C to standard_encoding_of(<N>).

c) Then we hardwire the following three inputs to the circuit in C to standard_encoding_of(,), i.e. 011.

d) Then we hardwire the following 3s inputs to the circuit in C to gadgets of the form:

0 hardwired to input of circuit in C
0 hardwired following input of circuit in C
input to circuit we are constructing hardwired to following input of circuit in C.

(this agrees with the representation of 0 and 1 in our definition of standard_encoding_of), where s is the number of inputs to the circuit that we are constructing. I now realise that I didn't get the corresponding construction in [2] right (on page 3): Sorry! It should have been:

y inputs are hardwired to gadgets of the form:
input to circuit we are constructing hardwired to input of circuit in C
input to circuit we are constructing also hardwired to following input of circuit in C
(this agrees with the defintion of encoding_of() in [2]).

e) Then we hardwire the following three inputs to the circuit in C to standard_encoding_of(,), i.e. 011.

f) Then we hardwire the following 3s inputs to the circuit in C to standard_encoding_of(1), i.e. 001, repeated s times.

g) Then we hardwire the last three inputs of the circuit in C to standard_encoding_of()) (standard_encoding_of the) symbol, that is), i.e. 100.

So, adding together the lengths of the sequences of inputs to the circuit in C and then taking to the power of k, the resulting circuit family should be of size big oh of:

```
                    (3 +   3r +   3 +   3s +   3 +   3s +   3)^k
(explanation:       ((a) + (b) +  (c) + (d) +  (e) + (f) +  (g))^k)
```

Adding together the four lots of 3 gives:

$$(12 + 3r + 3s + 3s)^k$$

Then adding together the two 3s terms gives:

$$(12 + 3r + 6s)^k$$

Now, if s (number of inputs to circuit that we are constructing) is sufficiently large then 6s will be greater that 3r (length of representation of control unit of Turing machine being simulated) so we will have $(12 + 3r + 6s)^k <= (12 + 2*6s)^k = (12 + 12s)^k$. And if s is sufficently large (>= 1 is sufficient here) then 12s will be greater than 12, so we will have $(12 + 12s)^k <= (24s)^k = 24^k*s^k$. And since 24^k is independent of s, the circuit family that we are constructing is of size big oh of s^k.

We assume P is monotonic (all positive coefficients should be sufficient to ensure this I think) so polynomial time bound of simulation should be at least P(s). So simulation should run to completion (not be terminated early by time bound) provided that P is big enough to accommodate running time of machine being stimulated for all possible values of s for the given value of k. This, in general, will require the coefficients and degree of P to depend upon k. I appreciate Fortnow implicitly encouraging me to clarify this with the questions that he asked me in his e-mails about [2].

So setting P big enough, for the given value of k, to accommodate the running time of my machine in [1] or Cai and Watanabe's sort-of equivalent machine for ALL input sizes (not just all sufficiently large input sizes) should give a circuit family for my [1] machine or Cai and Watanabe's sort-of equivalent machine. This contradicts the fact that these machines are constructed in such a way as to provably not have such a cct family: contradiction!

Conclusion:

So my construction still works if I change my conventions for the inputs to my enhancement to the universal Turing machine construction to the more standard ones, as suggested by Fortnow if I understand him correctly.

Conjectured Possible Next Step: Replace [2] Constructions Completely

This is on the back burner at the moment (metaphorically speaking, of course: not literally).

Acknowledgements:

1) I thank Professor Lance Fortnow, currently at Illinois Institute of Technology I believe, for his ongoing feedback (I have been corresponding with him about [2] by e-mail since I published it) on this.

2) I thank Dr Sione Ma'u, of the University of Auckland Mathematics Department, for his ongoing interest in my complexity theory research and ongoing logistical support with my communication about it with Professor Lance Fortnow.

3) I thank Professor Tava Olsen of the Melbourne Business School (in Australia: www.mbs.edu) for her ongoing interest in my Complexity Theory research, and her offer to try to read and understand my Complexity Theory research herself (although I think that she is not normally a Complexity Theory researcher herself).

References:

[1] https://blog.computationalcomplexity.org/2014/08/sixteen-years-in-making.html

[2] A Constructive Proof of Kannan's Theorem Based Upon a Much Simpler Construction: https://vixra.org/abs/2307.0031

[3] https://blog.computationalcomplexity.org/2002/12/foundations-of-complexity-lesson-11-np.html