A PROOF OF $\mathrm{P} \neq \mathrm{NP} \Leftrightarrow \mathrm{NP} = \mathrm{EXPTIME}$ USING MERKLE TREES

VICTOR PORTON

Abstract. My proof that $\mathrm{P} \neq \mathrm{NP} \Leftrightarrow \mathrm{NP} = \mathrm{EXPTIME}$.

I denote $s(X)$ the size of data X (in bits). I denote execution time of an algorithm A as $t(A, X)$.

I denote $f \sim g$ if two functions are both smaller than a polynomial of the other.

Theorem 1. $\mathrm{P} \neq \mathrm{NP} \Leftrightarrow \mathrm{NP} = \mathrm{EXPTIME}$.

Proof. First prove $\mathrm{P} \neq \mathrm{NP} \Rightarrow \mathrm{NP} = \mathrm{EXPTIME}$:

Assume $\mathrm{P} \neq \mathrm{NP}$.

We will need that there is a polynomial-time hashing algorithm h without hash collisions (in $\mathrm{P} \neq \mathrm{NP}$ assumption), because we use a Merkle tree. Article [1] proves that there are some polynomial-time algorithms (we can choose either SB and FSB) that are collision resistant (by reducing to an NP-complete algorithm).

Using a Merkle tree technology similar to one of the Cartesi [2] crypto (but with an infinite stack instead of a finite addressable memory and the size of hashes associated with nodes growing polynomially (we can take linear) regarding the input size; because memory is infinite, we adjust size of Merkle trees during execution; we also need to change the command system to accomodate the infinite memory).

Here is a model similar to Cartesi, but simplified (using this model, we prove the above paragraph):

- We have some deterministic stack machine with tree infinite stacks (input, output, and memory) of finite words addressed by *addresses* (natural numbers).
- We have also stack storing history of execution (see below).
- We can join memory, stack, and the history into one $\mathbb{N}$-addressable address space by interleaving them (for example, by putting n-th element into $4n$, $4n + 1$, $4n + 2$, $4n + 3$ addresses of the address space). The address space is considered to be a stack immersed into zero-initialized memory.
- We assume that CPU commands are one-word (if you like, you can consider an architecture with multi-word commands considered as several concatenated one-word commands) and the CPU has no command cache, to ensure a CPU command is always read from memory

before executing it, to have a complete "trace" of algorithm execution in the Merkle tree (see below).

- We require that one command executed grows the stacks no more than by const words. (This is necessary for the data described by the Merkle tree to have no more than $\text{const} \cdot t(A, x)$ elements. (*Remark:* It would be enough to allow instead the stacks to grow as any polynomial function of execution time.) *Remark:* We may allow the program to read/write any element of the stacks, not necessarily only its top.
- We have a provably (when P $\neq$ NP) collision-free hash function h. We will change our hash function value at $h(0, \ldots, 0)$ to be 0 (that essentially doesn't break being collision-free for purposes of a Merkle tree, because a Merkle tree implementation does not need to compare hashes of different levels; if it happens to have a collision with 0, it is easily fixable adding one to the values of hashes $h(x)$ when $x \neq 0$). It allows to optimize away "zero" Merkle tree nodes, so needing to create only one node when creating a Merkle tree.
- Hashes size will grow polynomially to the input data size (we consider below at-most exponential-time algorithms, so hashes grow at-most as $s(X) \sim \log t(A, X)$, what by the definition of provably collision-free hashes is enough for asymptotically zero probability of collision of the root hash used below for verification).
- We put program into the memory stack and input into the input stack.
- We create a Merkle tree of the address space stack (extended with zeros to be a degree of 2, what does not change algorithm's complexities, because the size is to be changed maximum 2 times). (*Remark:* For greater efficiency, we could instead create an initially one-node Merkle-like tree having two child trees "the input-output tree" (in turn having two child trees of one node with the value 0 as its hash value: "input" and "output") and "execution history" tree: with the value 0 as its hash value.)
- Every time a CPU command w is read (by the CPU) from an address a from memory, push to the execution history the value a together with metadata not to confuse different tuples of addresses ($s(a)$ grows at-most logarithmically with execution time) and update the Merkle tree (possibly replacing it with a new Merkle tree of more nodes as necessary, this is a constant time operation, because we don't bother to hash zeros).
- Every time a word is stored (by the CPU) into an address of output, update the Merkle tree (possibly replacing it with a new Merkle tree of more nodes as necessary, this is a constant time operation, because we don't bother to hash zeros).
- Start execution from the zeroth command in memory.
- The output is the output stack.

Remark: A practical implementation would have a different memory model to be more efficient.

Remark: Storing into Merkle tree is a logarithmic (to algorithm execution time) (even if multiplied by hash size) operation of $t(A, X)$ that is polynomial of $s(X)$ (we below consider only at-most exponential algorithms), so it does not change our algorithm complexity class.

So, our tree at the end of execution (non-deterministically) verifies both the input data, output data, and the entire sequence of execution steps (which includes every command executed) of our algorithm, so it verifies that it produces output data from input data by a certain sequence of commands. So, having the final tree, we can (non-deterministically) verify whether the output data is produced by a given input data by the given algorithm.

The size of the data and time to be used to verify the result of a decision algorithm A on input data X non-deterministically is proportional $\log t(A, X) \cdot \log t(A, X) \cdot \log t(A, X) = \log(3t(A, X))$. (The second multiplier is because the hash-size grows proportionally to the tree size (for at-most exponential algorithms) and the third one because addreses a pushed into execution history stack grow logarithmically to the execution time. We didn't take into account the time needed to push the input data, that does not matter for the complexity classes such as P, NP, or EXPTIME.

Take some exponential-time algorithm. It's execution time $t(A, X) \leq 2^{p(s(X))}$ (X is input data, p is a polynomial).

Therefore it can be non-deterministically verified (using Merkle trees) in time proportional to $\log(3 \cdot 2^{p(s(X))}) \sim s(X)$.

We have proved that every at-most exponential decision problem can be non-deterministically verified in at-most polynomial time.

In other words, a decision problem f for input X can be verified by an algorithm v that solves an NP problem using the formula $v(X, a) = 1$ where a is a data such that $s(a) \sim s(X)$.

The verification (because it's an NP-problem) in turn can be verified by a polynomial-time algorithm u using the formula

$$u(X, a, b) = 1$$

where b is a data such that $s(b) \sim s(X) + s(a) \sim s(X)$.

So, $f(X) = 1$ can be verified by u using the data (a, b) in polynomial time. Thus f is in NP.

We have proved P ≠ NP ⇒ EXPTIME = NP. The reverse implication is a common knowledge. □

References

[1] Augot, Daniel, Matthieu Finiasz, and Nicolas Sendrier. "A Fast Provably Secure Cryptographic Hash Function." International Association for Cryptologic Research. Accessed July 5, 2021. https://eprint.iacr.org/2003/230.pdf.

[2] Teixeira Augusto, and Diego Nehab. "The Core of Cartesi." Cartesi.io: Cartesi - Smart Contracts Taken to the Next Level. Accessed June 27, 2021. https://cartesi.io/cartesi_whitepaper.pdf.

Email address: porton@narod.ru