A PROOF OF P = NP USING MERKLE TREES

VICTOR PORTON

Abstract. My proof that P = NP. No efficient algorithm produced.

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

Assume P $\neq$ NP for hashes to work.

We will need that there is a provably polynomial-time algorithm without hash collisions (in P $\neq$ NP assumption), because we use a Merkle tree. FSB [1] is such a function.

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 logarithmically regarding the input size; because memory is infinite, we split memory not into chunks by halves but into interleaving fragments; we also need to change the instruction set to accomodate the stack kind of memory).

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

- We have some stack machine with two infinite stacks (disk and memory) of finite words addressed by *addresses*.
- The stacks are immersed into infinite memories that are initially zeros.
- We have an infinite binary tree with root $\mathbb{N}$ (considered as an ordered set), left subnode being the even-indexed elements of its parent, and right subnode being the odd-indexed elements of its parent. (In other words, memory is split into interleaving "blocks", "subblocks", etc.)
- We have some 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 $x \neq 0$).
- Hashes size (fixed) will grow proportionally to logarithm of the input data size (we consider below at-most exponential-time algorithms, so hashes grow at least as $s(X) \sim \log t(A, X)$ that is logarithm of the Merkle tree size, see below) size, 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 create an initially one-node Merkle 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 "state" tree: with the value 0 as its hash value.
- Initialize input tree to the Merkle tree of the input data (we need to pad it with zeros to be of degree of two size, what does not change algorithm's complexities, because the size is to be changed maximum 2 times).
- Every time a command w is read (by the CPU) from address a, change (or add together with its parents) to the Merkle tree the node that has a as its first element with the value $h(0, 0, w, a)$.
- Every time a word w is stored (by the CPU) into address a in memory, change (or add together with its parents) to the Merkle tree the node that has a as its first element with the value $h(0, 1, w, a)$.
- Every time a word w is stored (by the CPU) into address a in disk, change (or add together with its parents) to the Merkle tree the node that has a as its first element with the value $h(1, 1, w, a)$.
- As the first step push to the disk stack the program code (note: constant time) and data (by one word).
- As the second step start execution from the zeroth command.
- The output is the disk.
- Change the output tree to be the Merkle tree of the output data (padded with zeros to be of a degree of 2 size).

Remark: We need to plan our data not to change meaning when padded with zeros to be of a degree of 2 size. That's obviously possible without changing complexities.

Remark: Storing into Merkle tree is a logarithmic (to algorithm execution time) (even if multiplied by hash size, that is the logarithm of the logarithm) operation, 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 instruction executed and all operations that read input and produce output) of our algorithm, so it verifies that it produces output data from input data by a certain sequence of instructions. So, having the final tree, we can (non-deterministically) verify (given the same algorithm and so the same initial memory state) 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 X non-deterministically is proportional $\log t(A, X) \cdot \log t(A, X) \sim \log t(A, X)$. (The second multiplier is because the hash-size grows proportionally to the tree size (for at-most exponential algorithms). 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 EXP).

Take some algorithm in NP. It's execution time $t(A, X) \leq \text{const} \cdot 2^{s(X)}$ (X is input data size).

Therefore it can be non-deterministically verified in $\log(\text{const} \cdot 2^{s(X)} + \log(\text{const} \cdot 2^{s(X)})) \sim s(X)$ time.

A decision problem f can be made into the bijective function

$$x \mapsto (x, f(x), m(A, x))$$

where $m(A, x)$ is the Merkle tree of execution of algorithm A for our problem.

To have the domain and image of this decision problem well-defined and well-described, we can take f to be SAT.

So, an algorithm that computes a bijective function, which is exponential-time in both directions, is (for every bit of output) in NP (in both directions), because it is non-deterministic polynomial time (in both directions).

Therefore, it is polynomial-time (in both directions).

P = NP (the proof didn't produce an efficient algorithm).

Note that now we do know a polynomial-time (but not very efficient) NP-complete algorithm.

References

[1] Wikipedia contributors, "Fast syndrome-based hash," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Fast_syndrome-based_hash&oldid=949755869 (accessed June 29, 2021).

[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