On Computation and Random Strings

Sam Hopkins

June 3, 2011

Contents

1 Introduction

This paper will prove (and hopefully illuminate) a recent result in computational complexity theory [1]. Research in computational complexity seeks to understand the computational difficulty of algorithmically-solvable problems and, more generally, to understand the relationships between a variety of computational properties that those problems may satisfy. For example, (to jump right to the most important open question in complexity theory), we might ask, "if, for some problem L, we can easily computationally verify or reject any proposed solution to L, can we just as easily *find* a solution for L?"

The results presented here expose a connection between computational complexity theory and *algorithmic information theory*, otherwise (and henceforth here) known as *Kolmogorov complexity*. Broadly speaking, Kolmogorov complexity seeks to analyze the amount of information contained in an object (for example, a finite string) by considering the shortest description of that object: roughly, the Kolmogorov complexity of an object is the amount of information required to uniquely describe it.

We are interested here, however, in one very particular aspect of Kolmogorov complexity: its use to provide a mathematically-rigorous definition of randomness. The idea is this: a string (we may as well give up talk about "objects" at this point and move to slightly better-defined terms) is random if there are no patterns present in it. For example, *prima facie,* the string 0101010101 is very nonrandom, while the string 111010010100010 is much more so. But patterns can always be used to describe, so the Kolmogorov complexity of a patterned string will be small while the complexity of a pattern-less string will be large. Indeed, to describe a pattern-less string we will have to use the string itself as its own description. And here we have a definition of Kolmogorov-randomness: a finite string is random if it cannot be described by anything shorter than itself.

The results presented here characterize the computational power of the random strings. By "computational power" we mean (roughly) the power of a computer that knows which strings are random and which are not. As it turns out, the general problem of determining when a string is Kolmogorov-random is an uncomputable task, so the question is a slightly strange one. Nonetheless we will be able to prove that any problem that *does* admit computation and can be easily computed by a computer with access to the Kolmogorov-random strings requires only a certain amount (polynomially-much) space to compute.

2 Preliminaries

We have a lot of definitions to cover. The reader is welcome to skip this section on first reading and return as needed to understand the main exposition.

2.1 Strings and Languages

We first need a number of definitions from elementary theory of computation. The reader is referred to e.g. [4] for complete exposition. An *alphabet* Σ is

a finite set of characters. We usually take $\Sigma = \{0, 1\}$. A string over Σ is a finite sequence of elements of Σ; formally, it is a function $f : \{1, \cdots, n\} \to \Sigma$ for some $n \in \mathbb{N}$. We use absolute value bars for the length of strings; for example, $|001| = 3$. We denote a sequence of n 1s as 1^n (and similarly for other characters). So $1^5 = 11111$. We denote by Σ^* the set of all finite strings over an alphabet Σ. A language L over an alphabet Σ is a subset of Σ^*. We denote the empty string by ϵ.

The strings in Σ^* admit a natural ordering as follows: $\epsilon, 0, 1, 00, 01, 11 \cdots$. We call this ordering the *lexicographic ordering*.

2.2 Turing Machines

Now we need some computers. A *Turing Machine* (or TM) consists of

- a finite tape alphabet Σ,
- a two-way infinite *tape* partitioned into discrete squares into each of which may be put a symbol of Σ,
- A finite directed graph. We call the nodes of the graph *states*. The edges are labeled with a pair (r, w) where $r \in \Sigma$ and $w \in \Sigma \cup \{L, R\}$,
- a set A of nodes in the graph called *accepting states*.
- a tape head with a position on the tape
- a state q_0 designated to be the start state

In a *deterministic* TM, for any state q the set of edges $\{(r_i, w_i)\}$ exiting q must be such that $r_i \neq r_j$ for all i, j. A *nondeterministic* TM need not satisfy this requirement.

We think of a TM not as a static mathematical structure but as a computing machine. Given an input (a string written on the tape, with one character per square), the TM begins in state q_0 and proceeds by a series of state transitions wherein, beginning in some state q_i, it reads the symbol under the tape head and follows the edge (r, w) exiting q_i to state q_j while executing the instruction w. If $w \in \Sigma$, the TM erases the symbol under the tape head and replaces it with w; if $w = L$ then the tape head is moved one square to the left, and if $w = R$ then the tape head is moved one square to the right. If there is no such edge (r, w), the TM ceases to compute; we say that it *halts*. If it halts while in an accepting state, we say the TM accepts the input.

In any given state transition, a deterministic TM may have only one potential exit edge, but a nondeterministic TM may have several. We think of the computation of a nondeterministic TM as (potentially) branching at each state transition into multiple computation branches. We say that a nondeterministic TM accepts an input if there is some computation path that accepts.

2.3 Recognizers and Deciders

We say that a TM M recognizes a language L_M if for all $x \in \Sigma^*$, M accepts x if and only if $x \in L_M$. We take without proof the fact that the class of languages for which there exist recognizing TMs is invariant under the choice of deterministic or nondeterministic models.[1] A TM M that halts on all inputs is called a *decider*. We call a language L *recursively enumerable*, or r.e., if there exists a TM recognizing L. L is *recursive* if M is a decider.

An easy counting argument yields the result that nonrecursive languages exist. There are countably-many TMs, but there are $|\mathcal{P}(\Sigma^*)| = 2^{\aleph_0}$ possible languages over a finite alphabet. Thus there exist (uncountably-many!) nonrecursive languages. For the purposes of this paper, however, we are interested only in recursive languages. We will denote with REC the set of recursive languages (over $\Sigma = \{0, 1\}$).

2.4 Alternation

An *alternating Turing Machine* is a nondeterministic TM where each state is labeled either as *universal* or *existential* (think of universal or existential quantifiers). The computation of an alternating TM proceeds like that of a nondeterministic TM, but we will define string acceptance differently. Fix some input x and suppose we have a computation tree for an alternating TM A on x. We label leaf nodes as *accepting** if they correspond to accepting states in A. Then we recursively label the nodes in the tree as follows: if q corresponds to a universal state, label it *accepting** if every one of its children is *accepting**; if q corresponds to an existential state, label it *accepting** if one if its children is *accepting**. Finally, then, we say that A accepts x if the root node of the computation tree—the one corresponding to the start state—is *accepting**. The machines are so named for their ability to efficiently decide problems involving alternating universal and existential quantifiers.

2.5 Complexity Classes

We can characterize the computational difficulty of deciding membership in a recursive language in a number of ways; the reader is referred to [4] for a full complement of definitions. In this paper, we are interested in time and space complexity. Let M be a deterministic deciding TM. Time complexity of M is the function $f(n)$, where $f(n)$ is the largest number of steps in the computation of M on x for all x such that $|x| = n$. The space complexity of M is the function $f(n)$ giving the largest number of tape squares scanned by M on x for $|x| = n$.

[1] The proof is straightforward, though: clearly any language recognized by a deterministic TM D has a corresponding nondeterministic recognizer, since D is itself a nondeterministic TM. The other direction is less trivial: we construct a deterministic TM M to simulate a nondeterministic TM N. M will simulate all the computation paths of N on an input x and search for an accepting input. N has only finitely-many computation paths, so M will halt if N does, although M will take a *very* long time to do so.

The corresponding definitions for nondeterministic and alternating machines add one additional clause. The nondeterministic time complexity $f(n)$ of a nondeterministic or alternating TM N is the greatest number of steps in any branch of its computation (equivalently, the depth of the computation tree) on any input of length n. The space complexity is the greatest number of tape squares scanned by any branch of the computation on any input of length n.

We write $\mathsf{TIME}(t(n))$ to denote the set of languages L for which there exists a deterministic TM M deciding L with time complexity $O(t(n))$ and $\mathsf{SPACE}(s(n))$ for the set of languages decidable by a deterministic TM with space complexity $O(s(n))$. $\mathsf{NTIME}(t(n))$ and $\mathsf{NSPACE}(t(n))$ are the corresponding sets for nondeterministic machines; $\mathsf{ATIME}(t(n))$ corresponds to alternating TMs.

Example 1. The language $L = \{x : x \text{ contains a } 1\}$ is in $\mathsf{TIME}(n)$. Just let M be a machine that scans the input and halts when it reaches the end, halting in an accepting state if and only if it scanned a 1 somewhere along the way.

Now we are in a position to define some important complexity classes. We denote

$$\mathsf{P} = \bigcup_{k=0}^{\infty} \mathsf{TIME}(n^k)$$
$$\mathsf{NP} = \bigcup_{k=0}^{\infty} \mathsf{NTIME}(n^k)$$
$$\mathsf{AP} = \bigcup_{k=0}^{\infty} \mathsf{ATIME}(n^k)$$
$$\mathsf{PSPACE} = \bigcup_{k=0}^{\infty} \mathsf{SPACE}(n^k).$$

We take the following without proof.

Lemma 2.1. $\mathsf{PSPACE} = \mathsf{AP}$

Proof. See [4], theorem 10.21.

2.6 Computable Functions

We have TMs that recognize languages; we will also want to talk about TMs that compute functions. We say that a (deterministic) TM M computes a function $f : \Sigma^* \to \Sigma^*$ (not necessarily total) if for all $x \in \operatorname{dom} f$, when M is started with input x it halts with $f(x)$ (and nothing else) on its tape. We will sometimes be sloppy and confuse machines and functions, writing $M(x)$ for the output of M run on x and referring to $\operatorname{dom} M$ as the set of input strings for which M halts. A function is *computable* if there is a TM computing it. A function f is computable in $\mathcal{R}$ for some complexity class $\mathcal{R}$ if there is a TM M with complexity $\mathcal{R}$ computing f.

Example 2. The function $f(1^n) = 1^{2n}$ is computable. One algorithm might be to insert a 0 between any two 1s and a zero at the end of the input, then change all 0s to 1s and halt.

2.7 Boolean Circuits and Reductions

We will not give a totally formal definition of a boolean circuit, since it is not really necessary to understand the results that follow (and the reader will fairly easily be able to construct one graph-theoretically should she so desire). This definition is from [4].

Definition (Boolean Circuit)**.** A *Boolean circuit* is a tree and a set I of inputs. Each node in the tree is labeled either as an AND gate, an OR gate, or a NOT gate. Each node has exactly one parent. AND and OR gates have two children, and NOT gates have one child. A child can either be another node in the tree or can be an element in the set of inputs. Boolean circuits take input in the form of a function $f : I \to \{0, 1\}$ and propagate the values through the gates in the obvious way. The output of a circuit on an input is the value at the root node once the input has propagate all the way up to the top.

If λ is a circuit and $q_1, \cdots, q_n \in \{0, 1\}$ are inputs, we write $\lambda(q_1, \cdots, q_n)$ for the output of λ with the given inputs.

We frequently want to speak of reducing one language to another, since it gives us a way to analyze membership in complexity classes. If $A \in \mathsf{P}$ and B is such that, given $x \in \Sigma^*$ we can compute in polynomial time some function f such that $x \in A$ if and only if $f(x) \in B$, then clearly $B \in \mathsf{P}$. We call this type of reduction *mapping reduction*. In this paper, we will find a slightly different notion of reduction convenient to work with. Our definition is from [1]. We say that a language A *polynomial-time-truth-table reduces* to B if there is some function f computable in polynomial time such that $f(x) = (\lambda, q_1, \cdots, q_n)$ (suitably encoded into a string) such that $x \in A$ if and only if $\lambda(B(q_1), \cdots, B(q_n)) = 1$, where $B(q_i) = 1$ if $q_i \in B$ and $B(q_i) = 0$ otherwise. We write $A \leq_{tt}^{\mathsf{P}} B$. If we replace P by an arbitrary complexity class $\mathcal{R}$, we write $A \leq_{tt}^{\mathcal{R}} B$.

2.8 Enumeration

An *enumerator* for a set A is a Turing machine M computing a function $f : \{1^n : n \in \mathbb{N}\} \to A$. Essentially it computes an ordering on A. We take without proof the following easy lemma:

Lemma 2.2. *A language $A \subset \Sigma^*$ is r.e. if and only if there exists an enumerator for A.*

Proof. See [4], theorem 3.21.

3 Kolmogorov Complexity

3.1 Elementary Kolmogorov Complexity

3.1.1 Universal Turing Machines

A universal Turing Machine interprets its input string as an ordered pair (M, w), where M is a description of a TM and w a string, and accepts if and only if M accepts w, halting with only the output of M run on w on its tape. It is no surprise to us now that there exists a universal Turing Machine (in fact there are countably-many of them); a computer running an operating system is a universal Turing Machine (ignoring space bounds, of course). We will abuse notation when it is convenient, writing $U(M, x)$ and $U(y)$ interchangeably, as it will sometimes be convenient to thing of universal TMs as "just" functions on strings. Here y will be thought of as a binary encoding of the pair (M, x) (which must of course be encoded into binary to be fed into the TM at all).

3.1.2 Plain Kolmogorov Complexity

We are finally in a position to define our first notion of Kolmogorov complexity (though it will not be our final one). Let U be a universal TM. We define the function $C_U : \Sigma^* \to \mathbb{N}$ as follows:

$$C_U(x) = \min\{|y| : U(y) = x\}.$$

$C_U(x)$ is thus the length of the shortest available description of x. We now show that C_U does not depend strongly on the choice of U.

Lemma 3.1. *Let U and U' be universal TMs. Then there is a constant c such that for all $x \in \Sigma^*$*

$$C_U(x) \leq C_{U'}(x) + c.$$

Proof. Let M be a description of U'. Then for any x, if $U'(y) = x$ then $U(M, y) = x$. Thus the constant c is just the length of the description of M. □

We are now justified in ignoring differences between various C_Us, so we fix a universal TM U and let $C = C_U$. Unfortunately, while delightfully simple, C has some failings as a description of complexity. In particular, we would like to have complexity behave nicely with respect to concatenation: namely, we would like to have $C(xy) \leq C(x) + C(y) + c$ for some constant c. Unfortunately, this turns out not to be true. As it happens,

Theorem 3.2. *For any fixed $d \in \mathbb{N}$, there exist $x, y \in \Sigma^*$ such that $C(xy) \geq C(x) + C(y) + d$.*

Proof. See [2], theorem 3.1.4.

In intuitive terms, the issue is this: concatenation introduces extra computational difficulty (without, intuitively, introducing extra complexity) because

it is computationally difficult to find the break between strings. Indeed, if we *tell* the TM where the break in the strings is, we can obtain an upper bound on $C(xy)$.

Theorem 3.3. *For $x, y \in \Sigma^*$, $C(xy) \leq 2C(x) + C(y) + c$.*

Proof. Let $\overline{x}$ denote x with all characters doubled. For instance, if $x = 101$, then $\overline{x} = 110011$. Let M be a machine that takes input of the form $\overline{x}01y$ and outputs xy (M simply removes doubled characters until it reaches the sequence 01. It removes the 01, then halts). Then to describe a string xy we can do the following. Let x' be the shortest string such that $U(x') = x$ and y' the shortest such that $U(y') = y$. Then of course $|x'| = C(x)$ and $|y'| = C(y)$. As input to U we use the string $\overline{x}'01y'$ and an encoding of a TM N which first simulates M on its input, then simulates U on each of the output strings and concatenates the result.[2] The length of the encodings of all these TMs is constant, so the input has length $2|x'| + |y'| + c = 2C(x) + C(y) + c$ and the proof is complete. □

This is only one of several problems plaguing plain Kolmogorov complexity (see [2] section 3.5). The issues are alleviated by employing a somewhat modified definition.

3.2 Prefix-Free Kolmogorov Complexity

Our troubles (or at least the ones used here to motivate the definitional change) are with concatenation. If we ensure that concatenated strings are unambiguously decomposable into their component substrings, our problems should be dissolved. To this effect we define a *prefix-free* set as follows: A set $A \subset \Sigma^*$ is prefix free if for all $x \in A$ there is no y in A such that $xy \in A$. More pithily: no string is a proper prefix of another.

A function $f : \Sigma^* \to \Sigma^*$ is prefix-free just in case dom f is, and a TM M is prefix-free just in case the set of inputs on which M halts is prefix-free.

Intuitively, the density of string lengths in a prefix-free set must be low: once some long string x is in a prefix-free set A, none of its initial substrings may be in A. The following inequality makes this formal. The proof is adapted slightly from [3].

Theorem 3.4 (Kraft's Inequality)**.** *Let l_n be a finite or infinite sequence of natural numbers. There is a prefix-free set $A \subset \Sigma^*$ with strings of length l_n (i.e. A and the l_ns can be put into 1-to-1 correspondence with each string x corresponding to some $l_i = |x|$) if and only if*

$$\sum_{n=1}^{\infty} 2^{-l_n} \leq 1.$$

[2] Things aren't quite this simple – we will actually need to modify M slightly so that on output the strings x and y remain distinguishable.

Proof. Suppose first that we have a prefix-free set A; we show that $\sum 2^{-|x|} \leq 1$. We associate with each string $x \in A$ an interval $\Gamma_x \subseteq [0,1)$. Let

$$\Gamma_x = [0.x, 0.x + 2^{-|x|})$$

where $0.x$ denotes the real number with binary decimal expansion given by the digits in x. We claim that for $x \in A$, the Γ_x are disjoint. Suppose otherwise. Then there are $x, y \in A$ such that $0.x \leq 0.y \leq 0.x + 2^{-|x|}$. But then $0.x$ and $0.y$ can differ only at some decimal place $n > |x|$, so x is a proper prefix for y, a contradiction.

In the other direction, suppose we have a set of lengths l_n with $\sum 2^{-l_n} \leq 1$. Let Γ_n be a sequence of disjoint intervals in $[0,1)$ with lengths $2^{-l_1}, 2^{-l_2}, \cdots$ (whose existence is of course guaranteed by hypothesis). Then take strings x_n such that $0.x_n$ is the left endpoint of Γ_n. Since the Γ_n are disjoint, an analogous argument to the above shows that the set $\{x_n\}$ is prefix-free. □

We define a *universal prefix-free Turing Machine* as follows: a prefix-free TM U is universal prefix-free if, for any prefix-free machine M, there is a string z_M such that for all $x \in \operatorname{dom} M$, $U(z_M, x) = M(x)$. That is, U is itself prefix-free and U can simulate any prefix-free machine. We take without proof the existence of such a machine.

Now we can define prefix-free Kolmogorov complexity. Let U be a universal prefix-free TM. Let $K_U(x) = C_U(x)$ (where C_U is the usual Kolmogorov complexity function). As before (and by a similar argument), the choice of universal TM affects K by at most a constant, so we fix a universal prefix-free TM U and let $K = K_U$. (We will, however, want to discuss complexity functions for particular universal prefix-free machines later on.) Note that we can also talk about Kolmogorov complexity functions for arbitrary prefix-free machines, although of course the resulting functions may be much less well-behaved under perturbations of the input. We denote such a function for a machine M by K_M.

3.3 Randomness and Kolmogorov Complexity

Intuitively speaking, a string is random if it fails to obey some sort of pattern. We can now formalize this notion, with the following motivation. Suppose a string x follows a pattern. Then we may describe x by describing the pattern rather than by directly describing x itself, and if the pattern is sufficiently simple, the length of the resulting description will be less than $|x|$. So we define a string to be random if it does not admit a description with length less than its own. Formally, we say x is *random* if $K(x) \geq |x|$. We write R_{K_U} for the set of strings random relative to a particular function K_U and R_K for the Kolmogorov random strings in general.

A simple counting argument shows that random strings of every length exist.[3] The proof is from [4].

[3]The author feels fairly certain that in the case of prefix-free complexity a stronger argument could be made to demonstrate the existence of many more random strings than the theorem here shows, since the domain of possible descriptions is so severely restricted, but he does not know of such.

Lemma 3.5. *For all $n \in \mathbb{N}$ there exists a random string of length n.*

Proof. There are 2^n strings of length n but only $2^n - 1$ strings of length $m < n$. Since a string can describe at most one other string, there is at least one string of length n with no description of length less than n. □

Before we move on, we note one more fact about random strings: they are not computable.

Theorem 3.6. $R_K \notin \mathsf{REC}$.

Proof. Suppose R_K is recursive and let M decide it. Make a new machine M' that does the following: on input $k \in \mathbb{N}$, output the lexicographically-least string $x \in R_K$ of length at least k. Let c be the size of a description of M'. Then we have a description of x of length $\log k + c$; that is, we know $K(x) = |x| \leq \log k + c \leq \log |x| + c$. But for large $|x|$ this is a contradiction. □

3.4 Overgraphs

We define the overgraph of a function $f : \Sigma^* \to \mathbb{N}$, denoted $\operatorname{ov} f$, as follows:

$$\operatorname{ov} f = \{(x, y) : f(x) \leq y\}.$$

Lemma 3.7. $\operatorname{ov} K$ *is r.e..*

Proof. Let M interpret its input as an ordered pair (x, y). We want M to accept if and only if $K(x) \leq y$. M iterates over all strings of length less than or equal to that of y and checks each string z for $U(z) = x$. If it finds one, it accepts. □

3.5 Existence Lemma for Prefix-Free TMs

We take the following lemma without proof.

Lemma 3.8 ([1])**.** *Let $f : \Sigma^* \to \mathbb{N}$ be such that $\sum_{x \in \Sigma^*} 2^{-f(x)} \leq 1$ and $\operatorname{ov} f$ is r.e.. Then there is a prefix-free Turing Machine M such that $f(x) = K_M(x) - 2$.*

Proof. See [1], Theorem 2.1.

4 Computation and Random Strings

We are finally in a position to state our main result. The remainder of the paper will be devoted to its proof.

Theorem 4.1 (Allender, Friedman, Gasarch, in [1])**.** *Let $\mathcal{U}$ denote the set of universal prefix-free TMs. Then*

$$\mathsf{REC} \cap \bigcap_{U \in \mathcal{U}} \{A : A \leq_{tt}^{\mathsf{P}} R_{K_U}\} \subseteq \mathsf{PSPACE}.$$

To rephrase: the set of recursive languages polynomial-time truth-table reducible to R_{K_U} for every $U \in \mathcal{U}$ is a subset of PSPACE.

4.1 A Sufficient Condition for Theorem 4.1

By the following, it actually suffices to prove

$$\mathsf{REC} \cap \bigcap_{U \in \mathcal{U}} \{A :\leq_{tt}^{\mathsf{P}} \operatorname{ov} K_U\} \subseteq \mathsf{PSPACE}.$$

Lemma 4.2.

$$\mathsf{REC} \cap \bigcap_{U \in \mathcal{U}} \{A : A \leq_{tt}^{\mathsf{P}} R_{K_U}\} \subseteq \mathsf{REC} \cap \bigcap_{U \in \mathcal{U}} \{A : A \leq_{tt}^{\mathsf{P}} \operatorname{ov} K_U\}.$$

Proof. It suffices to show that if $A \leq_{tt}^{\mathsf{P}} R_{K_U}$ then $A \leq_{tt}^{\mathsf{P}} \operatorname{ov} K_U$. Fix A such that $A \leq_{tt}^{\mathsf{P}} R_{K_U}$. Then by definition there is some q computable in polynomial time such that $q(x) = (\lambda, q_1, \cdots, q_n)$ where $\lambda(R_{K_U}(q_1), \cdots, R_{K_U}(q_n)) = 1$ if and only if $x \in A$ (where we conflate R_{K_U} and its characteristic function). But clearly $q_i \in R_{K_U}$ if and only if $(q_i, |q_i| - 1) \notin \operatorname{ov} K_U$. So if M computes q, let M' simulate q, then translate each output q_i into $(q_i, |q_i| - 1)$ and add an extra NOT gate to λ. Clearly if M runs in polynomial time then M' does as well, since there can only be polynomially-many q_i. The function q' computed by M' witnesses $A \leq_{tt}^{\mathsf{P}} \operatorname{ov} K_U$. □

Our strategy will be to proceed by contradiction to show that if $L \notin \mathsf{PSPACE}$ for $L \in \mathsf{REC}$, then there is a universal prefix-free TM such that $L \not\leq_{tt}^{\mathsf{P}} \operatorname{ov} K_U$. Fix $L \notin \mathsf{PSPACE}$ with $L \in \mathsf{REC}$. We will painstakingly construct a function F and define with it a function H such that

1. F is total and $\operatorname{ov} F$ is r.e..
2. $H(x) = \min(K(x) + 5, F(x) + 3)$.
3. $\sum_{x \in \Sigma^*} 2^{-H(x)} \leq \frac{1}{8}$.
4. $L \not\leq_{tt}^{\mathsf{P}} \operatorname{ov} H$.

By the following, this suffices to prove the result.

Lemma 4.3. *If H satisfies (1)-(4), then there exists $U' \in \mathcal{U}$ such that $H = K'_U$.*

Proof. By property (2), $F(x) + 3 \geq H(x)$, so

$$\sum_{x \in \Sigma^*} 2^{-F(x)-3} \leq \sum_{x \in \Sigma^*} 2^{-H(x)} \leq \frac{1}{8}.$$

But (suppressing notation) we have

$$\sum 2^{-F(x)-3} = \frac{1}{8} \sum 2^{-F(x)}$$

and so

$$\sum 2^{-F(x)} \leq 1.$$

Then since by hypothesis $\operatorname{ov} F$ is r.e., by Lemma 3.8 we have that $F(x)+2 = K_M$ for some prefix-free machine M.

Now we need a couple of propositions:

Proposition 4.4. Let $U \in \mathcal{U}$ and $c \in \mathbb{N}$. Then there is $U' \in \mathcal{U}$ with $K_{U'}(x) = K_U(x) + c$.

Proof. Let U' be a machine that, when started on input x, removes the string 0^c from the beginning of x, then simulates U. If x does not begin with 0^c, then U' enters an infinite loop. Then $\operatorname{dom} U' = \{0^c x : x \in \operatorname{dom} U\}$. I claim that $\operatorname{dom} U'$ is prefix-free. Suppose otherwise and let $x, y \in \operatorname{dom} U'$ such that x is a proper prefix for y. We know $x = 0^c x'$ and $y = 0^c y'$ for some $x', y' \in \operatorname{dom} U$. But then clearly x' is a proper prefix for y', contrary to hypothesis. Additionally, U' is universal; it just requires its input (the language of TM/input pairs) to be coded slightly differently, with a prefix of 0^c added to each string coded to be readable by U. Finally, clearly for all x we have $K'_U(x) = K_U(x) + c$. □

Proposition 4.5. Let $U, \in \mathcal{U}$ and let M be a prefix-free machine. Then there is $U' \in \mathcal{U}$ such that $K_{U'}(x) = \min(K_U(x), K_M) + 1$.

Proof. Let U' be as follows. U' begins by checking the first character of its input. If the character is 1, U' erases the 1 and simulates U on the remainder of the input. If the character is 0, U' erases the character and simulates M on the input. Then we have

$$\operatorname{dom} U' = \{1x : x \in \operatorname{dom} U\} \cup \{0x : x \in \operatorname{dom} U'\}.$$

By the same argument as in Proposition 4.4, each of $A = \{1x : x \in \operatorname{dom} U\}$ and $B = \{0x : x \in \operatorname{dom} U'\}$ is prefix-free, and no element of either can be a proper prefix for an element in the other, since the initial characters don't match. Thus $\operatorname{dom} U''$ is prefix-free. Also by the same argument as above, U' is universal (since U is, and we can simply ignore the behavior of U' on strings beginning with a 0 by considering them to be invalidly coded), and clearly $K_{U'}(x) = \min(K_U(x), K_M(x)) + 1$ which is what we wanted to show. □

Returning to the proof of the lemma: by Proposition 4.4 there exists $U' \in \mathcal{U}$ such that $K_{U'}(x) = K(x) + 4$. Then by Proposition 4.5 we have

$$\begin{aligned} H(x) &= \min(K(x) + 5, F(x) + 3) \\ &= \min(K(x) + 4, F(x) + 2) + 1 \\ &= \min(K_{U'}(x), K_M(x)) + 1 \\ &= K_{U''}(x) \end{aligned}$$

for some prefix-free machine $U'' \in \mathcal{U}$. □

4.2 Constructing H

4.2.1 Overview

It now remains to construct such an H for our fixed $L \notin \mathsf{PSPACE}$. We will construct our function F recursively. We first give a somewhat informal discussion of the process.

At each stage i of the recursion, we will have a current version of F, denoted F_i. Furthermore, we will build up the function K alongside F. K, of course, is already defined, but we will enumerate elements into its graph (or, more precisely, into its overgraph) at each stage. We denote by K_i the version of K at stage i. We let $H_i(x) = \min(K_i(x) + 5, F_{i-1}(x) + 3)$. H is always defined, since F_j will be total for all j. The $i - 1$ is present because, at each stage, we will update K, which updates H, and then we will choose what to add to F dependant on H (so F won't "catch up" until the next stage).

Let $F_0(x) = 2|x| + 3$ and let $K_0 = \varnothing$. Fix some computable enumeration κ of $\operatorname{ov} K$. Then we let $\operatorname{ov} K_{i+1} = \operatorname{ov} K_i \cup \kappa(i)$, and let $K_{i+1}(x) = \min\{y : (x, y) \in \operatorname{ov} K_{i+1}\}$. The idea will be to play a game against κ. At each stage, κ enumerates something into $\operatorname{ov} K$ and we will respond by enumerating the appropriate thing into $\operatorname{ov} F$ to ensure that $L \not\leq_{tt}^{\mathsf{P}} \operatorname{ov} H$. So we let $\operatorname{ov} F_i = \operatorname{ov} F_{i-1} \cup \{(z_j, r_j)\}$, where each (z_j, r_j) is a yet-to-be-determined pair (we may add multiple pairs to $\operatorname{ov} F$ at a given stage, although only a single pair will be added to $\operatorname{ov} K$). Then we let $F_i(x) = \min\{y : (x, y) \in \operatorname{ov} F_i\}$. We will eventually take $\operatorname{ov} F = \bigcup \operatorname{ov} F_i$ and $F(x) = \min\{y : (x, y) \in \operatorname{ov} F\}$. A little lemma is called for at this point.

Lemma 4.6. *Let $i < j$. Then $F(x) \leq F_j(x) \leq F_i(x)$ and $H(x) \leq H_j(x) \leq H_i(x)$.*

Proof. Exercise. □

Let $\gamma_1, \gamma_2, \cdots$ be a computable enumeration of the possible polynomial-time truth-table reductions of L to $\operatorname{ov} H$. The enumeration can be computed as follows. Let γ_i^* just be the ith string in the lexicographic ordering that codes a TM (under some fixed coding of TMs). Modify the ith machine by placing a "clock" in it. That is, we let γ_i be a machine that simulates γ_i^* but increments a counter each time γ_i^* takes a step in its computation. The clock runs out at n^i, where i is the input size, at which point γ_i halts, ceasing to simulate γ_i^*. Then for any function f, if f is computable in time n^k, there is some n^{k+j}-time machine γ_i computing f (where j is just whatever is the necessary extra time to run the clock). If f is computable but not in time n^k for any k, however, then for every γ_i computing f, the clock n^i will run out (for at least one input) before f has halted, so there will be no γ_i computing f.

We interpret the output of each γ_i as a Boolean circuit λ_i and a set of queries $\{(z_j, r_j)\}$ whose purpose in life is to be tested for membership in $\operatorname{ov} H$. If the output of $\gamma_i(x)$ cannot be so interpreted, we consider it to be undefined.

It suffices satisfy the requirement

$$R_e : \gamma_e \text{ is not a polynomial-time-truth-table reduction of } L \text{ to } \operatorname{ov} H.$$

for all $e \geq 1$. At step e we will start trying to satisfy R_e. In order to satisfy R_e, we will need to find a witness x such that

$$x \in L \Leftrightarrow \lambda_e[\operatorname{ov} H(z_1, r_1), \cdots, \operatorname{ov} H(z_n, r_n)] = 0$$

where we conflate $\operatorname{ov} H$ and its characteristic function.

Let us foster some intuition here. Our control over $\operatorname{ov} H$ comes from our control over F. We can always ensure, for any fixed pair (z, r), that $(z, r) \in \operatorname{ov} H$ by putting $(z, r-3)$ into $\operatorname{ov} F$: this ensures that $F(z) \leq r-3$ and so $H(z) \leq r$. But if we make $H(z)$ too small, it may fail to satisfy $\sum 2^{-H(x)} \leq 1/8$. For each R_e, we will find a witness x as above and play a game $\mathcal{G}_{e,x}$ against κ. At recursive step i we will potentially make a move any game $\mathcal{G}_{e,x}$ for $e \leq i$. If we make a move on some $\mathcal{G}_{e,x}$ for some $e < i$, however, we will destroy all games $\mathcal{G}_{e',x'}$ with $e < e' \leq i$ and construct new ones. In this way, each game $\mathcal{G}$ can be played as though all the games below it have finished—if such turns out not to be true we rebuild $\mathcal{G}$ at each recursive step until all games below it are actually finished.

4.2.2 The Game $\mathcal{G}_{e,x}$

We now give the rules for $\mathcal{G}_{e,x}$. Let $\gamma_e(x) = \{\lambda, (z_1, r_1), \cdots, (z_n, r_n)\}$. Let

$$X_e^* = \{z_i : \text{there is some } r_i \text{ such that } (z_i, r_i) \in \gamma_e(x)\}.$$

X_e^* is the set of strings that form the first half of each of the queries generated by $\gamma_e(x)$. We don't quite want to work with this whole set, though. First of all, we remove all the queries whose results are already known. Suppose this game is generated at stage i. Then if $H_i(z) \leq r$, we know $H(z) \leq r$ (by Lemma 4.6). Then replace the inputs in λ corresponding to any (z_j, r_j) such that $H_i(z_j) \leq r_j$ with TRUE and simplify λ accordingly—denote this new circuit λ'. (Importantly, this can be done by a TM: recall that we must end up with $\operatorname{ov} F$ being r.e..) Then any z_j corresponding to a removed input can be removed from X_e^*; let X_e^{**} be the resulting set.

Finally, in playing game $\mathcal{G}_{e,x}$, we don't want to affect any game $\mathcal{G}_{e',x'}$ for any $e' < e$. So let

$$X_e = X_e^{**} \setminus \bigcup_{e'<e} X_{e'}.$$

The following proposition will be useful later.

Proposition 4.7. For all $z \in \Sigma^*$, there is at most one e such that $z \in X_e$.

Proof. Suppose $z \in X_e$. Then clearly $z \notin X_{e'}$ for any $e' < e$, because then by construction $z \notin X_e$, and by construction $z \notin X_{e'}$ for any $e' > e$. □

We play the game on a DAG (directed acyclic graph) D with labeled nodes. Each node is labeled with two things:

1. A function $h : X_e \to \mathbb{N}$. The function h is a possible value for H given the information about H contained in H_i. Formally speaking, $h(z) \leq H_i(z)$ for all $z \in X_e$.

2. A projected value VAL of the output of λ if $h \equiv H$. That is, $\text{VAL} = \lambda'(\operatorname{ov} h(z_1), \cdots, \operatorname{ov} h(z_n))$, where we conflate $\operatorname{ov} h$ and its characteristic function.

Let there be a node in D for each such function h. We now give a bound on the size of D.

Lemma 4.8. *We can uniquely code any node in D in a string of length at most $|x|^{2e}+1$.*

Proof. Since γ_e has a clock on it running out after n^e steps (where, recall, n is the input length), the set X_e contains at most $|x|^e$ elements. Now, since $H \leq F_0 + 3$ (by Lemma 4.6), for any z we have $H(z) \leq 2|z| + 3 + 3 = 2|z| + 6$. Since it is of course positive, $H(z)$ can take on at most $2|z| + 6$ values for any given input z. Furthermore, for each $z_j \in X_e$, we have $|z_j| \leq |x|^e$ (since z_j was output by γ_e). Thus there are $(2|x|^e + 6)^{|x|^e}$ choices for h. For large $|x|$, the dominating term of the expansion of this bound is $(2|x|^e)^{|x|^e}$, which is dominated for large $|x|$ by $2^{|x|^{2e}}$. We can represent this number in $\log 2^{|x|^{2e}} \approx |x|^{2e} + 1$ bits, so given some (computable) ordering on the possible choices for h, we can specify a particular node in a string of length most $|x|^{2e} + 1$. □

Now we will finish describing D. We designate one node as the *start node*. It is the node where $h \equiv H_i$. Let (h, VAL) and (h', VAL') be nodes in D. There is an edge from (h, VAL) to (h', VAL') just in case $h'(z) \leq h(z)$ for all $z \in X_e$. The idea is that the nodes are connected just if for some j we could have $h \equiv H_j$ and then for some $k > j$ we could have $H_k \equiv h'$.

Proposition 4.9. D is acyclic.

Proof. Exercise (*hint: use Lemma 4.6*).

The game is played between two players, YES and NO. The game begins with a single token, placed on the start node. Each player has a score, each starting at 0, where high scores are bad. A player's score, in the end, will represent how much he has contributed to the difference between h and H_i. Player YES goes first, after which the players alternate turns. A turn consists of moving the token from a node (h, VAL) to a node (h', VAL'). After the move,

$$\sum_{z \in X_e} 2^{-h'(z)} - 2^{-h(z)}$$

is added to the player's score. A move is legal if the following two conditions are satisfied

1. There is an edge from (h, VAL) to (h', VAL').
2. The player's score after the move will not exceed 2^{-e-i_e-6}, where i_e is the number of times we have had to rebuild $\mathcal{G}_{e,x}$ as a result of moves being played on games $\mathcal{G}_{e',x}$ for $e' < e$ (details will be given later).

The game ends when the player whose turn it is cannot make a legal move. At that point, the player YES has won if the token is on a node where $VAL = 1$ and the player NO has won if the token is on a node where $VAL = 0$. That is, YES wins if the result of $\mathcal{G}_{e,x}$ says that $x \in L$ and NO wins otherwise.

Proposition 4.10. The game $\mathcal{G}_{e,x}$ always ends after finitely-many moves.

Proof. Exercise.

We make a note here that will be relevant later. We have spoken of $\mathcal{G}_{e,x}$ ending, and we will speak of winning strategies for $\mathcal{G}_{e,x}$, but it is important to remember, when we give the details of the construction of F, that we will not always play the games all the way through. Sometimes they will get destroyed, and sometimes we will simply cease to play them.

In game-theoretic terms, this is a deterministic game of perfect information. The players both know all the moves available and there is no luck involved. We claim that therefore either YES or NO always has a winning strategy.

Proposition 4.11. YES or NO always has a winning strategy.

Proof. Suppose that YES plays each turn so as to prevent NO from winning on his next turn. If YES can always succeed in doing this, then YES had a winning strategy to begin with (NO will not win, so YES must). If there is some sequence of moves resulting in a situation where YES cannot do this, then NO wins on the following turn. NO therefore had a winning strategy to begin with: he merely must play the sequence of moves to force the losing situation for YES. □

If YES has a winning strategy, we denote $\operatorname{val} \mathcal{G}_{e,x} = 1$. Otherwise, let $\operatorname{val} \mathcal{G}_{e,x} = 0$. We take without proof (though it is not at all hard to see) that the function val is computable.

4.2.3 Details of the Recursion

To recall our goal: we want functions F and H satisfying the conditions in 4.1. In particular, $\operatorname{ov} F$ must be r.e..

The reader is referred to section 4.2.1 for definitions of functions during the recursion. We need just one more for now: at stage 0, for all e set $i_e = 0$. Also, let LEX be a computational object that, when called, returns strings in the lexicographic order. (Formally we could make LEX a function $lex : \mathbb{N} \to \Sigma^*$ and suppose that as we recurse we store a number n_i; then to call LEX amounts to taking $swag(n_i)$ and incrementing $n_{i+i} = n_i + 1$. But because we want to think of this recursion as being computed rather than as a static mathematical construction, it is easier to think of calling LEX than to deal with the formalities of storing n_i.)

Now for the real work. Here is what happens at stage s for $s \geq 1$. Begin by updating K_i and therefore automatically H_i. There is a little subtlety to how to update H_i, however: H_i must only be updated on inputs z with $z \in X_e$ for some $e \leq s$. This doesn't change the eventual values of H (since of course every X_e is eventually in play), but it will be important later when we obtain some bounds. Consider next all the requirements R_e for $1 \leq e \leq s$. At each one of two possibilities obtains: either there is or there is not a pre-existing game associated with R_e.

No Pre-Existing Game There may be no game $\mathcal{G}_{e,x}$ yet being played, either because it was destroyed at stage $s-1$ or because this is the first time we have considered R_e (i.e. if $e = s$). We then call LEX until we get a string x with the property that if we define a new game $\mathcal{G}_{e,x}$ using H_s, then it will be the case that $\operatorname{val} \mathcal{G}_{e,x} \neq L(x)$ (where we conflate L and its characteristic function). Of course, it is not obvious that such a string even exists and that we can get it from finitely-many calls to LEX. But not to worry: here is the result.

Lemma 4.12. *Such an x can always be found in at most finitely-many calls to LEX.*

Proof. Suppose for purposes of contradiction that at some stage s we find that there is no such x corresponding to a requirement R_e. We will show that this gives a PSPACE algorithm deciding L, contrary to hypothesis. By Lemma 2.1, it suffices to construct a polynomial-time alternating TM M deciding L.

First of all, hard-code into M all values of L (i.e. the characteristic function of L) on strings preceding x in the lexicographic ordering. On input z, the machine M computes as follows. If z precedes x in the lexicographic ordering, check the hard-coded lookup table and accept if $L(y) = 1$ and not otherwise. If z does not precede x, the machine M will construct a game $\mathcal{G}_{e,x}$ using H_s and accept just in case $\operatorname{val} \mathcal{G}_{e,x} = 1$. By hypothesis, this machine will decide L.

It remains to show that M can compute $\operatorname{val} \mathcal{G}_{e,x}$ from H_s in polynomial time. The idea is that computing the existence of a winning strategy for YES amounts to examining a sequence of nested alternating quantifiers: does there exist a move for YES such that for all moves for NO there exists a move for YES such that for all moves for NO there exists ... such that YES wins. This, of course, is what alternating TMs are designed to do.

The number of nodes in the DAG D for game $\mathcal{G}_{e,x}$ is exponential in $|x|$, so it will not do to have M actually construct all of $\mathcal{G}_{e,x}$ on any given computation path (remember that the computation paths nondeterministically branch). Recall, though, that by Lemma 4.8 it takes at most $|x|^{2e} + 1$ bits to represent a particular node in D. But the specification of some node, the scores of YES and NO, the number i_e, and $\lambda_{e,x}$ is a complete description of the game. Thus it is easy in polynomial time (even on a standard TM) to nondeterministically choose a modification h' of h where h' is such that there is an edge in D from (h, VAL) to (h', VAL') and check to see if a move of the token from specified node to the node (h', VAL') is legal for either player.

We will not give a formal alternating polynomial time algorithm to compute $\operatorname{val} G_{e,x}$, but the idea is as follows. The algorithm is recursive, with the recursion beginning with the first turn of the game. At each step of the recursion, we store the game state (as in the preceding paragraph) using polynomially-much space. Suppose we are looking for a winning strategy for YES. Then at some recursive step where the token is on a node (h, VAL), the algorithm does the following:

1. Check the scores. If YES has won, accept. If YES has lost, reject.
2. Divide the nodes (h', VAL') such that there is an edge from (h, VAL) to (h', VAL') into two equinumerous classes (for example by the values of

h' on certain inputs) and nondeterministically select each class, repeating recursively until there is just one node left in the class. If this is YES's turn, the branching should be universal (because we want YES to be able to win given any move that NO makes); if this is NO's turn the branching should be existential.

3. Return to (1) with new node (h', VAL').

It is clear that the algorithm accepts only if YES has a winning strategy. Furthermore, I claim that the algorithm runs in polynomial time. We know we can check the scores of the players in polynomial time (since we may as well just pass those down with the recursion). At each step in the branching, there are exponentially-many nodes to branch to, but by doing this divide-and-conquer trick, no individual computation path will see more than polynomially-many splits until the splitting process finishes and the computation path moves on to checking another node.

Finally, we claim that the length of all paths in D is uniformly bounded by some polynomial in $|x|$. We will construct the longest possible path in D. We will choose a path such that for any adjacent (h, VAL) and (h', VAL') in the path, h and h' differ by just one on just one input. For convenience, we will suppose that $h_1(z)$ (i.e. the first h in the path) is equal to $|z|$; carrying out the proof with the additional constant factors is straightforward. For each $z_i \in X_e$, there are $|z_i|$ nodes in the path (since for each z_i we will need a step to decrement $h(z_i)$ from $|z_i|$ to 0). There are at most $|x|^e$ elements in X_e, and for all i, we have $|z_i| \leq |x|^e$. So

$$|z_1| + \cdots + |z_{|x|^e}| \leq (|x|^e)^2$$

which is polynomial in $|x|$. Thus, our algorithm to calculate $\operatorname{val} \mathcal{G}_{e,x}$ runs in alternating polynomial time and is therefore in PSPACE, which is a contradiction. So after finitely-many calls to LEX we will get the witness we want. □

Returning to the main construction, once such an x is found, we begin the game $\mathcal{G}_{e,x}$ (though we do not play a move yet), playing as YES if $\operatorname{val} \mathcal{G}_{e,x} = 1$ and NO otherwise (i.e. we play as the player with the winning strategy).

Pre-Existing Game There are three subcases.

No Change It may be the case that the new element enumerated into $\operatorname{ov} K$ by κ at this stage was not in X_e, in which case no move has been played on $\mathcal{G}_{e,x}$. In this case we leave the game alone.

κ Has Cheated It may be the case that whatever was enumerated into $\operatorname{ov} K$ resulted in a move on $\mathcal{G}_{e,x}$ which caused a player to make an illegal move—that is, to make a move resulting in a score greater than 2^{-e-i_e-5}. In this case we destroy game $\mathcal{G}_{e,x}$.

It's Our Turn It may be the case that it's our turn to play on $\mathcal{G}_{e,x}$, either because the game was started at the last stage and we are the YES player, or because κ played a legal move on the last stage. In this case, we play a move according to our winning strategy. We effect the move by enumerating elements into $\operatorname{ov} F$, which changes H_s and moves the token from wherever it was to some new node.

If either of the latter two cases occurs, we say that R_e is acting, and we destroy all games $\mathcal{G}_{e',x}$ with $s \geq e' \geq e+1$ and increment all such $i_{e'}$ by one. If R_e has acted, we immediately proceed to the next stage. If not, we continue on to $e+1$.

This completes the details of the recursion. It is really best thought of as an algorithm, since it is at once a construction of H and a process by which $\operatorname{ov} F$ can be enumerated. We will not formally prove that the algorithm we have just described can be performed by a TM, but it should be fairly obvious (especially since we do not require it to happen in any particular time or space bound).

4.3 The Punchline

It may not seem it, but we are almost done. We would first like to show that our functions F and H are well-defined. (Of course, H is well defined as long as F is.) The following lemma is the key.

Lemma 4.13. *For all e, there are at most finitely-many stages where R_e is acting. Furthermore, after those stages, R_e is satisfied.*

Proof. We proceed by strong induction on e. Suppose the claim holds for all $e' < e$. Then there is a stage s after which all of $R_{e'}$ cease to act (since there are finitely-many e').

Let $\mathcal{G}_{e,x}$ be R_e's game at stage s. If $\mathcal{G}_{e,x}$ is never destroyed at a later stage, then eventually R_e must cease to act, since $\mathcal{G}_{e,x}$ will end in finitely-many moves (either the game will end by one player winning, or κ will simply cease to enumerate things that affect X_e). The same argument holds if $\mathcal{G}_{e,x}$ is destroyed only finitely-many times.

So suppose that $\mathcal{G}_{e,x}$ is destroyed infinitely-many times. We will derive a contradiction. We know that for all $e' < e$, the requirements $R_{e'}$ cease to act after s, so past s, the counter i_e never changes. Furthermore, the only way $\mathcal{G}_{e,x}$ can be destroyed past s is for κ to play a move such that its score exceeds $\epsilon = 2^{-e-i_e-6}$. Fix one such game.

Suppose we assign indices i, j to the nodes in in D in the order that the token in $\mathcal{G}_{e,x}$ lands on them until κ "cheats". Let i range over nodes on which the token landed as a result of our move, and let j range over nodes on which the token landed as a result of κ's move. Then we have

$$\sum_{i,j \in D} \left(\sum_{z_k \in X_e} 2^{-h_j(z_k)} - 2^{-h_i(z_k)} \right) \geq \epsilon.$$

These moves were κ's, though. The only way for κ to effect change on h is by enumerating elements into ov K. So, letting t be the stage where $\mathcal{G}_{e,x}$ was begun and letting K_t and K_s be the function K at the respective stages, we have

$$\sum_{z_k \in X_e} 2^{-K_s(z_k)} - 2^{-K_t(z_k)} \geq \epsilon.$$

But if this happens at infinitely-many stages, then

$$\sum_{z_k \in \Sigma^*} 2^{-K(z_k)} \geq \epsilon + \epsilon + \epsilon + \cdots = \infty.$$

This is impossible, since ran K can be put into one-to-one correspondence with a prefix-free set, so by Kraft's inequality (Theorem 3.4), we have a contradiction.

It remains now to show that R_e is satisfied. We have shown that R_e ceases to act after some stage s. After that stage, we have won the game $\mathcal{G}_{e,x}$. If val $\mathcal{G}_{e,x}$ was 0, then we played as NO and x was in L. Furthermore, by the construction of X_e at each step, we know that at no stage after s does ov H_s restricted to X_e change. Thus, if the token ends up on a node $(h, VAL) \in D$, we have $VAL = 0$ and therefore $\lambda_{e,x}(\text{ov } H(z_1), \cdots, \text{ov } H(z_k)) = 0$.

The argument is symmetric for the case val $\mathcal{G}_{e,x} = 1$. This concludes the proof. □

It is now easy to show that the functions F and H are well-defined. Suppose we wish to know the value of $F(z)$ for some $z \in \Sigma^*$. By Proposition 4.7, there is at most one X_e with $z \in X_e$. By Lemma 4.13, R_e eventually stops acting, so there is some i past which all the ov F_j for $j > i$ agree on elements of the form (z, r). But then clearly $F = \bigcup F_i = \lim_{i \to \infty} F_i$ is well-defined. If z never appears in X_e, recall, we have $F(z) = 2|z| + 3$. From the well-definedness of F of course it follows that H is well-defined. A similar argument invoking Proposition 4.7 and Lemma 4.13 shows that $H(x) = \min(K(x) + 5, F(x) + 3)$. Essentially, Proposition 4.7 and Lemma 4.13 together show that this construction is well-behaved with respect to limits.

Let us take stock of the situation. We have functions F and H such that F is total, ov F is r.e., $H(x) = \min(K(x) + 5, F(x) + 3)$, and $L \not\leq_{tt}^{\mathsf{P}} \text{ov } H$. It remains to show that $\sum_{x \in \Sigma^*} 2^{-H(x)} \leq \frac{1}{8}$; then by section 4.1 we will get Theorem 4.1.

Lemma 4.14. $\sum_{x \in \Sigma^*} 2^{-H(x)} \leq \frac{1}{8}$.

Proof. For any particular z, we know $H_0(z) = F_0(z) = 2|z| + 6$. We will analyze $2^{-H(z)}$ by examining how much $H(z)$ has changed over the course of the recursive construction. We therefore break up $2^{-H(z)}$ like this:

$$2^{-H(z)} = 2^{-H_0(z)} + (2^{-H(z)} - 2^{-H_0(z)}).$$

Everything is positive, so we can sum over $z \in \Sigma^*$ and manipulate sums as we please to get

$$\sum_{z \in \Sigma^*} 2^{-H(z)} = \sum_{z \in \Sigma^*} 2^{-H_0(z)} + \sum_{z \in \Sigma^*} 2^{-H(z)} - 2^{-H_0(z)}.$$

Bounding the first term is easy. We know $H_0(z) = 2|z| + 6$, and we know that there are 2^n strings of length n, so we have

$$\sum_{z\in\Sigma^*} 2^{-H_0(z)} = \sum_{z\in\Sigma^*} 2^{-2|z|-6} = \frac{1}{64}\sum_{n=0}^{\infty}(2^n)(2^{-2n}) = \frac{1}{32}.$$

So now we need to evaluate the second term. We want to obtain a relationship between the term and the scores of κ and F (i.e. our score) on various games. To begin with, we will further telescope out the sum:

$$\sum_{z\in\Sigma^*} 2^{-H(z)} - 2^{-H_0(z)} = \sum_{s\geq 0}\sum_{z\in\Sigma^*} 2^{-H_{s+1}(z)} - 2^{-H_s(z)}.$$

But $H_{s+1}(z)$ and $H_s(z)$ differ only on inputs z with $z \in X_e$ for some $e \leq s+1$ (by the way we've done the updating of H at each stage). So this expression is equal to

$$\sum_{(s\geq 0)}\sum_{(e\leq s+1)}\sum_{(z\in X_e)} 2^{-H_{s+1}(z)} - 2^{-H_s(z)}.$$

But since each z appears in only one X_e, we have

$$\sum_{z\in X_e} 2^{-H_{s+1}(z)} - 2^{-H_s(z)} = \sum_{z\in X_e} 2^{-h'(z)} - 2^{-h(z)}$$

where we suppose that the moves on game $\mathcal{G}_{e,x}$ at stage s has resulted in the token shifting from node h to node h'. But this is just the combined score increase of F and κ on game $\mathcal{G}_{e,x}$ at stage s. Now we introduce some notation. Let $\mathbf{G}$ denote the set of all games played at any stage of the recursion. Let $\mathcal{S}(\mathcal{G})$ denote the sum of κ's and F's score at the end of game $\mathcal{G}$ (i.e. either when $\mathcal{G}$ is destroyed or when the last move is played on it). Then we have proved the following:

Proposition 4.15.

$$\sum_{z\in\Sigma^*} 2^{-H(z)} - 2^{-H_0(z)} = \sum_{\mathcal{G}\in\mathbf{G}} \mathcal{S}(\mathcal{G}).$$

We will first consider κ's scores. Here a very loose bound will suffice. Let us suppose that κ's score is much greater than it likely is; let us in fact suppose that $H(z) = K(z) + 5$ for all z and that κ has made all the moves resulting in this, picking up, in the process, an accumulated score of at most $\sum 2^{-K(z)-5}$. By Kraft's inequality, this is less than $\frac{1}{32}$.

Now consider the final score of F on games where κ cheats. We know that it is less, then, than κ's final score. But by the above, the sum over *all* of κ's scores, cheating and not cheating, is bounded by $\frac{1}{32}$, so the sum of F's scores in these cases is also bounded by $\frac{1}{32}$.

Finally, we need to consider the final score of F on games that are never destroyed or that are destroyed because a move gets played on another game.

On these games we know that F never cheats. Now, fix e and consider all games $\mathcal{G}_{e,x}$. On the first of these, F cannot contribute more than 2^{-e-6} to $\sum_{z\in\Sigma^*} 2^{-H(z)} - 2^{-H_0(z)}$. On the second, F cannot contribute more than 2^{-e-7}, since i_e has been incremented. In general, F cannot contribute more than

$$\sum_{i=6}^{\infty} 2^{-e-i} = 2^{-e} \sum_{i=6}^{\infty} 2^{-i} = \left(\frac{1}{32}\right) 2^{-e}.$$

on games $\mathcal{G}_{e,x}$ for fixed e. But then summing over all e, we see that F's scores cannot contribute more than $\frac{1}{32}$ in total, since

$$\sum_{e=1}^{\infty} 2^{-e} = 1.$$

Thus,

$$\sum_{z\in\Sigma^*} 2^{-H(z)} \leq \frac{1}{32} + \frac{1}{32} + \frac{1}{32} + \frac{1}{32} = \frac{1}{8}.$$

□

This completes the proof of our main result.

References

[1] Eric Allender, Luke Friedman, and William Gasarch. Limits on the computational power of random strings (forthcoming).

[2] Rodney Downey and Denis Hirschfeldt. *Algorithmic Randomness and Complexity.* Springer-Verlag, 2010.

[3] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag, 1993.

[4] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, 2nd edition, 2005.