Kolmogorov Complexity and the Symmetry of Algorithmic Information

James Bailie

November 1, 2016

1 Introduction

The symmetry of algorithmic information theorem (SAI) states that, up to an additive constant, the prefix Kolmogorov complexity $K(x, y)$ of x and y is equal to the sum of the complexity $K(x)$ of x and the complexity $K(y|x, K(x))$ of y given x and $K(x)$:

$$K(x, y) = K(x) + K(y|x, K(x)).$$

This theorem is the analogue to a result in probability theory: $\log P(x, y) = \log P(x) + \log P(y|x)$ where P is a probability measure. Moreover, there is obvious similarity between SAI and the result $H(X, Y) = H(X) + H(Y|X)$ where H is the Shannon entropy. Therefore, regardless of its applications, SAI deserves attention.

The proofs of SAI in the literature (see [3] and theorem 3.9.1, [6]) are either flawed, too condensed or do not give the necessary background. The motivation for this report is thus to provide a concise, self-contained proof of SAI. The report also includes all the necessary background to understand SAI and its proof.

The rest of this section will give an informal description of SAI. The definitions and reasoning given below will be formalised in the coming sections. Firstly, we introduce Kolmogorov complexity. Intuitively, we can measure how complex an object x is by the simplest description of x. For example, the string '1111111111' has a simple description 'ten copies of 1', while the string '0110111000' probably does not have such a simple description. So the former string is less complex than the latter.

Formalising the idea of a 'description' in computer science terms, we can think of a description of x as a programs that outputs x. So the complexity of x is simply the length of the shortest program p that outputs x:

$$C(x) = \min\{l(p) : p \text{ outputs } x\}.$$

Before we can properly define Kolmogorov complexity, we need to formalise the notion of a program. This will be done in the next two sections. For now, we will provide some more intuition about SAI.

A program can receive input. This input can provide useful information to help the program produce its output. If y is very similar to x, then there would probably be a short program that outputted y when given x as input. For example, if x was the first 1000 digits of π and y was the first 999 digits of π, the program would just have to output the first 999 digits of x and it would be done. However, if y is completely different to x, then providing x as input would not help very much. The shortest program that computed y given x as input would basically be the same as the shortest program that computed y with no input.

Analogous to the above definition, the conditional complexity of y given x is the length of the shortest program p that outputs y when given x as input:

$$C(y|x) = \min\{l(p) : p \text{ outputs } y \text{ given } x\}.$$

$C(y|x)$ measures the additional complexity of y, not present in x.

We can view a program p as outputting a pair (x, y), provided we have some encoding of the pair. Using this, we will make one more informal definition: the complexity of a pair (x, y) is the length of the shortest program p that outputs the pair (x, y):

$$C(x, y) = \min\{l(p) : p \text{ outputs } (x, y)\}.$$

Notice that if y is very similar to x, it would not take much more work to get (x, y) than it would to get just x. That is, if $C(y|x)$ is small, then $C(x, y) \approx C(x)$. Moreover, as y changes to become less similar to x or, equivalently, $C(y|x)$ gets bigger, we would expect that the difference between $C(x, y)$ and $C(x)$ increases. Therefore, we might conjecture $C(x, y) \approx C(x) + C(y|x)$. Unfortunately, it is not so simple - SAI requires $C(x)$ be given as input as well as x in $C(y|x, C(x))$. Then the relationship $C(x, y) = C(x) + C(y|x, K(x))$ holds.

We make the convention that all logarithms have base 2, unless explicitly stated otherwise.

My thanks to Marcus Hutter and Stephen Roberts for supervising this project and for their helpful advice. In particular, special thanks to Marcus for the proof of lemma 6.

2 Turing Machines

When we say that the complexity of an object x is equal to the shortest program p that outputs x, we need to specify what computer is running the program p. For this, we will use the standard model of computation, the Turing Machine, since it has several nice theoretical properties which will come in handy in formulating and proving results about Kolmogorov Complexity.

For our purposes, we will make the following formal definition of a prefix Turing machine. See figure 1. This non-standard definition will lead to a better formulation of Kolmogorov complexity in the section 4.

Definiton 1. *A **Turing machine** is an 7-tuple $\langle Q, I, O, W, \delta, q_0, F\rangle$ consisting of:*

- *a finite set of states Q,*
- *a one-sided, unidirectional, read only, input tape I,*
- *a one-sided, unidirectional, write only, output tape O,*
- *a two-sided, bidirectional, working tape W,*
- *a transition function $\delta : (Q \setminus F) \times \{0, 1\}^2 \to Q \times \{0, 1\}^2 \times \{N, R\}^2 \times \{L, R\}$,*
- *an initial state $q_0 \in Q$, and*
- *a set of final states $F \subset Q$.*

Each tape is divided into cells with each cell containing either 0 or 1. Associated with each tape is a tape head. At any point, the tape head is over a particular cell. It can read and write symbols onto the cell it is over and it can move to immediately adjacent cells.

All three tapes contain an infinite number of cells. The input and output tapes have a leftmost cell and extend infinitely to the right. In contrast, the working tape extends infinitely to the right and the left. Usually, some string is placed at the start of the input tape. If this string is finite then the rest of the input tape is filled with zeroes. We consider this string the input of the Turing machine. Cells of the other two tapes are initially filled with zeroes.

The head on the input tape can not write symbols, only read them, while the head on the output tape can not read symbols, only write them.

The transition function δ describes how the Turing machine runs. At any point, the Turing machine can make an action depending on

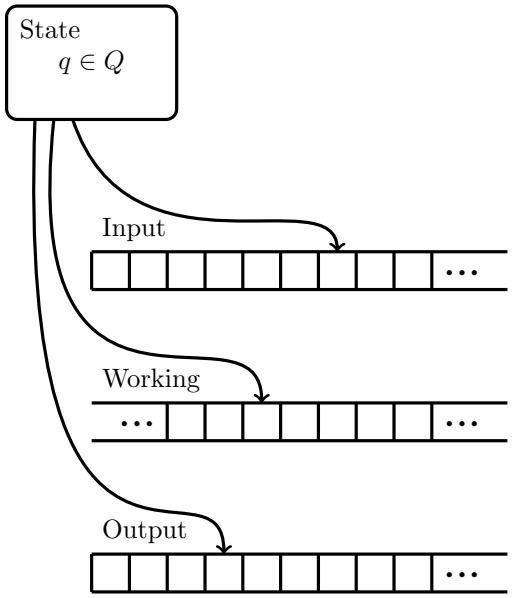

Figure 1: A Turing machine.

- the current state, $q \in Q$, provided $q \notin F$. (If $q \in F$ then the Turing machine makes no more actions and we say the Turing machine has halted.)
- the binary symbol in the cell under the input tape head, and
- the binary symbol in the cell under the working tape head.

Possible actions that a Turing machine can take are:

- transition to a new state,
- write a binary symbol in the cell under the working tape head,
- write a binary symbol in the cell under the output tape head,
- move the tape head to the right on the input or the output tapes, and
- move the tape head to the left or the right on the working tape.

Formally, we describe δ as follows: for each state $q_r \in Q \setminus F$ and symbols $b_r \in \{0,1\}, b'_r \in \{0,1\}$ on the input and working tapes, the transition function $\delta(q_r, b_r, b'_r) = (q_w, b_w, b'_w, d_w, d'_w, d''_w)$ specifies that the Turing machine should:

1. transition into the new state q_w,
2. write b_w in the working tape,
3. write b'_w in the output tape,
4. move the input tape right if $d_w = R$ or not move if $d_w = N$,
5. move the working tape left or right for d'_w equal to L or R, and,

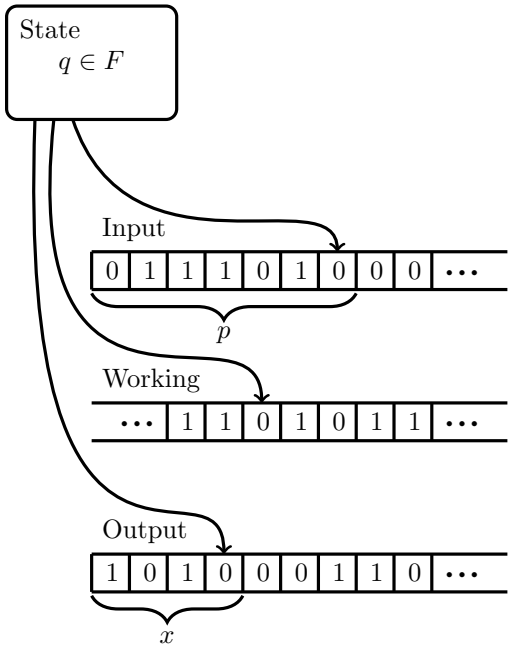

Figure 2: A Turing machine's input p and output x.

6. move the output tape right if $d''_w = R$ or not move if $d''_w = N$.

A Turing machine begins in the initial state q_0 with input and output tape heads over the left most cells and makes actions according to the transition function δ. Each of these actions is called a transition and takes one (time) step of computation. If the Turing machine transitions into a final state $q \in F$, it halts and stops computing. Otherwise, it continues making actions indefinitely, never stopping. We write $T(p) = x$ to mean that the Turing machine T outputs x when given the program p as input. More formally, $T(p) = x$ means that T halts with the binary string p on the input tape to the left of the head and the binary string x on the output tape to the left of the head. If T never halts when given p as input, we write $T(p) = \infty$. See figure 2.

Definiton 2. *The set of* ***programs*** *of a Turing machine T is*

$$\{p : T(p) = x \text{ for some } x \text{ or } T(p) = \infty\}.$$

So a binary string p is a program of T if T halts with p on the input tape to the left of the head or if T never halts and the input head reaches but does not move past p. Recall that since the input head is unidirectional, the Turing machine can never read further down the input tape past p and then return the input head to the end of p.

Using this construction, we can give a formal definition of the prefix Kolmogorov complexity of a binary string x, relative to a Turing machine T as:

$$K_T(x) = \min\{l(p) : T(p) = x\},$$

where $l(p)$ is the length of p.

It is a well known result that there are universal Turing machines that can simulate all other Turing machines. We will briefly explain what we mean by this but see [4, chapter 9] for a more complete exposition. Notice that a complete description $\langle Q, I, O, W, \delta, q_0, F\rangle$ of a Turing machine is finite. So we can encode a

description of a Turing machine as a finite binary string. Moreover, we can order these strings, say in lexicographic order (see 3 for details). This gives an enumeration $T_1, T_2, T_3, T_4, ...$ of all Turing machines. Then, a universal Turing machine U takes as input a tuple (i, p) and simulates p on T_i:

$$U(i, p) = T_i(p).$$

(See section 3 on how to encode a tuple for input into a Turing machine.) From herein, we will fix a universal Turing machine U and think of it as *the* universal Turing machine.

3 Prefix Codes

First, we need to define some notation for binary strings. Let $\mathbb{N} = \{0, 1, 2, ...\}$ be the set of natural numbers. Let $\mathcal{B} = \{0, 1\}$ and $\mathcal{B}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, ...\}$ be the set of finite binary strings. Let $x, y \in \mathcal{B}^*$. We write $x = x_1x_2x_3...x_m$ to mean that x starts with $x_1 \in \mathcal{B}$ followed by $x_2 \in \mathcal{B}$, then $x_3 \in \mathcal{B}$ and so on up to $x_m \in \mathcal{B}$. Define the function $l : \mathcal{B}^* \to \mathbb{N}$ that sends x to its length $l(x) = m$.

Given $x = x_1...x_m \in \mathcal{B}^*$ and $y = y_1...y_n \in \mathcal{B}^*$, define the concatenation $xy = x_1...x_my_1...y_n \in \mathcal{B}^*$ of x and y as the string comprising of x followed by y. Finally, define x^p for $p \in \mathbb{N}$ inductively: set $x^0 = \varepsilon$ and $x^p = xx^{p-1}$ to be the concatenation of x and x^{p-1} for $p > 0$.

We can order $\mathcal{B}^*$ in lexicographic order: first by length and then ordering 0 before 1 at each string index. That is to say, given two different binary strings $a = a_1a_2...a_m$ and $b = b_1b_2...b_n$ of the same length, a is ordered before b if

1. $l(a) < l(b)$ or
2. $l(a) = l(b)$ and $a_i < b_i$ for the first index i where a_i and b_i differ.

The first 15 binary strings in lexicographic order are:

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111.$$

Define the bijective function $\mathcal{N} : \mathbb{N} \to \mathcal{B}^*$ that sends n to the nth binary string in lexicographic order. Notice that $f(n)$ can be computed for any n by a Turing machine T in the following straightforward way: T increments through the lexicographic ordering of $\mathcal{B}^*$ until it reaches the nth string, which it outputs. In this case, we say $\mathcal{B}^*$ is effectively enumerable. See section definition 11 in 5 for details.

Importantly, using this function $\mathcal{N}$, we consider $\mathbb{N}$ and $\mathcal{B}^*$ as essentially the same. We define a Turing machine T with input $n \in \mathbb{N}$ as $T(n) = T(\mathcal{N}(n))$. More generally, for any function $g : \mathbb{N} \to \mathbb{N}$, we can think of the corresponding function $\mathcal{N}^{-1} \circ g \circ \mathcal{N} : \mathcal{B}^* \to \mathcal{B}^*$. Similarly, for functions $\mathcal{B}^* \to \mathcal{B}^*$. So we will no longer make a distinction between $\mathbb{N}$ and $\mathcal{B}^*$ and interchange between them freely.

Definiton 3. *Let $\mathcal{C}$ be a set of codewords. (Usually, $\mathcal{C} \subset \mathcal{B}^*$.) A **code** is a (partial) function $E : \mathbb{N} \to \mathcal{C}$ from natural numbers to codewords.*

Think of E as the encoding function. Note that E is a code for some (possibly strict) subset of $\mathbb{N}$. Some numbers $n \in \mathbb{N}$ may have zero codewords but we have restricted the definition of a code so that each n cannot have more than one codeword. Since we have a one-to-one correspondence between $\mathbb{N}$ and $\mathcal{B}^*$, a code is often specified as a function from $\mathcal{B}^*$.

Definiton 4. *Let $x, y \in B^*$. x is a **proper prefix** of y if there exists $z \in B^*$ such that the concatenation xz of x and z is equal to y.*

*A set $S \subset \mathcal{B}^*$ is **prefix free** if no string in S is a proper prefix of another string in S.*

*A code $E : \mathbb{N} \to \mathcal{C} \subset \mathcal{B}^*$ is **prefix free** if its set of codewords $\mathcal{C}$ is prefix free. In this case, we say E is a **prefix code**.*

The important property of a (known) prefix free set S is that for any two elements $x, y \in S$, it is always possible to uniquely determine x and y from their concatenation xy. Why is this? Suppose $xy = wz$ for $w, z \in S$. Then either x is a prefix of w, or w is a prefix of x, or $w = x$. But by assumption S is prefix free, so the first two cases are not possible. Thus, $x = w$ and consequently $y = z$. Prefix free sets are sometimes called self-delimiting sets for this reason.

We will now give a simple prefix code $\mathcal{E}$ for $\mathcal{B}^*$. Define $\mathcal{E}$ by $\mathcal{E}(x) = 1^{l(x)}0x$. Note $\mathcal{C} = \{1^{l(x)}0x : x \in \mathcal{B}^*\}$ is the set of codewords. We will use this prefix code to define a tuple. The tuple (x, y) is the concatenation $\mathcal{E}(x)y$. Note this is uniquely decodable, in the sense that we can obtain x and y from (x, y), since $\mathcal{E}$ is prefix free. Using this idea, it is possible to inductively define an n-tuple for any $n \in \mathbb{N}$. We only need a 3-tuple, which is defined as $(x, y, z) = (x, (y, z)) = \mathcal{E}(x)\mathcal{E}(y)z$.

We can now make sense of a Turing machine taking a tuple as input. When we write $T(x, y)$, we really mean $T(\mathcal{E}(x)y)$.

We can now explain why we defined a Turing machine in a non-standard way.

Proposition 1. *Given a Turing machine T, the set of programs*

$$\{p \in \mathcal{B}^* : T(p) = x \text{ for some } x \text{ or } T(p) = \infty\}$$

of T form a prefix free set.

Proof. Suppose $p, q \in \mathcal{B}^*$ such that p is a proper prefix of q. Suppose additionally that p is a program of T. Now consider T running with q on the input tape. T will continue to transition until it reads all of p. Since p is a program of T, the input head will never move past p. So T will never read all of q and therefore, q cannot be a program of T. □

Note that this proposition would not hold if we were using the standard Turing machine definition. This proposition turns out to have some very useful consequences, some of which we will show below.

Theorem 1 (Kraft's inequality)**.** *Let $A = \{a_1, a_2, ...\} \subset \mathcal{B}^*$ be a countable prefix free set of binary strings. Let l_i be the length of a_i. Then*

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1.$$

Conversely, given lengths $l_1, l_2...$ satisfying the above inequality, there exists a prefix free set $A \subset \mathcal{B}^*$ of binary strings, each of length $l_1, l_2, ...$

Note that the case where A is finite is a simple corollary of this theorem.

Proof. We need the concept of a cylinder set. For $x \in \mathcal{B}^*$, consider the binary expansion $0.x$. For example, if $x = 101$ then $0.x = 0.101 = \frac{5}{8}$. Define the cylinder set of x as the interval $\Gamma_x = [0.x, 0.x + 2^{-l(x)})$. (The terminology cylinder set comes from the fact that $\Gamma_x = \{0.\omega : \omega \in \mathcal{B}^* \text{ and } x \text{ is a prefix of } \omega\}$.)

We make the following observations:

1. If y is not prefix of x and x is not a prefix of y then $\Gamma_y \cap \Gamma_x = \emptyset$.
2. If y is a prefix of x then $\Gamma_x \subset \Gamma_y$.

We will now prove the first part of the theorem. Since A is prefix free, all the cylinder sets Γ_{a_i} of A are disjoint. Also, the intervals Γ_{a_i} are contained in the interval $[0, 1]$. It follows that the sum $\sum_{i=1}^{n} 2^{-l_i}$ of the lengths of the cylinder sets is no greater than 1.

To prove the second part, assume $l_1 \leq l_2 \leq ... \leq l_n \leq ....$ Choose disjoint adjacent intervals $I_1, I_2, ..., I_n, ...$ of length $2^{-l_1}, 2^{-l_2}, ..., 2^{-l_n}, ...$ respectively, starting at the left of $[0, 1)$. Since we assumed that the lengths were increasing, each interval I_n is of the form $[2^{-p_n}, 2^{-p_n} + 2^{-l_n})$ for some $p_n \in \mathbb{N}$. Now 2^{-p_n} has a unique infinite binary expansion, if we exclude the possibility of expansions ending in an infinite number of ones. Let a_n be the first l_n digits in the infinite binary expansion of 2^{-p_n}.

Since each of the intervals I_n are disjoint, $A = \{a_1, a_2, ..., a_n, ...\}$ is a prefix free set by observation 2. above. □

There are some important corollaries of Kraft's inequality. See corollary 2 in section 4. These corollaries are central to the idea behind SAI and would not be true if we had not made our particular non-standard definition of a Turing machine.

4 Kolmogorov Complexity

As in section 2, we define complexity as follows:

Definiton 5. *The **(prefix) Kolmogorov complexity** of $x \in \mathcal{B}^*$ relative to the Turing machine T is*

$$K_T(x) = \min\{l(p) : p \in \mathcal{B}^*, T(p) = x\}.$$

*The **(prefix) Kolmogorov complexity** of $n \in \mathbb{N}$ is the Kolmogorov complexity of the corresponding string $\mathcal{N}(n) \in \mathcal{B}^*$*

$$K_T(n) = K_T(\mathcal{N}(n)).$$

In section 2, we fixed a particular universal Turing machine U. We make the convention that *the* Kolmogorov complexity $K(x) = K_U(x)$ is defined as the Kolmogorov complexity relative to U. The invariance theorem below shows that it does not really matter which universal Turing machine we fixed.

Theorem 2 (the invariance theorem)**.** *Let U be any universal Turing machine. Then, for any Turing machine T, there exists a constant c_T such that*

$$K_U(x) \leq K_T(x) + c_T.$$

In particular, the constant c_T does not depend on x.

Proof. Let p be a program such that $T(p) = x$ and $l(p) = K_T(x)$. Let i be the index of T in our ordering of Turing machines. Then $U(i, p) = T(p) = x$. Thus, $K_U(x) \leq l(i, p) = 2l(i) + 1 + l(p) = 2l(i) + 1 + K_T(x)$. □

Corollary 1. *For universal Turing machines U and U', we have $K_U(x) = K_{U'}(x)$ for all x, up to an additive constant independent of x (but dependent upon U and U').*

The proof is trivial. Note that the additive constant can depend on U and U'. The point here is that if we chose to fix a different universal Turing machine U', all our results about Kolmogorov complexity would still hold, up to an additive constant.

We will also formalise the definitions of the complexity of a pair and the conditional complexity mentioned in the introduction.

Definiton 6. *The **(prefix) Kolmogorov complexity** $K_T(x, y)$ of a pair (x, y) is the complexity $K_T(\mathcal{E}(x)y)$ of the string $\mathcal{E}(x)y$:*

$$K_T(x, y) = K_T(\mathcal{E}(x)y).$$

In the introduction, we talked about a program receiving input. But now that we have the formal definitions in place, we talk about Turing machines having programs as input. To avoid confusing, we will call input to a program by the term 'side information'. So we say a Turing machine T receives input p and side information y and write $T(y, p)$ in this case. The universal Turing machine U may receive input p, side information y and index i and we write $U(y, i, p)$ in this case. $U(y, i, p)$ simulates T_i on input p with side information y.

Definiton 7. *The **(prefix) conditional Kolmogorov complexity** $K_T(x|y)$ of x given y is the length of the shortest program p that outputs x when given y as side information:*

$$K_T(x|y) = \min\{l(p) : p \in \mathcal{B}^*, T(y,p) = x\}.$$

Again, *the* Kolmogorov complexity of a pair and *the* conditional complexity is simply the complexity relative to our fixed universal Turing machine U:

$$K(x,y) = K_U(x,y) \text{ and } K(x|y) = K_U(x|y).$$

The complexity of a pair (n,m) of natural numbers and the conditional complexity of n given m are defined in the obvious way.

Corollary 2 (of Kraft's inequality)**.** *We have the following inequalities:*

1. $\sum_{x\in\mathbb{N}} 2^{-K(x)} \leq 1$,
2. $\sum_{y\in\mathbb{N}} 2^{-K(y|x)} \leq 1$ *for fixed* $x \in \mathbb{N}$,
3. $\sum_{y\in\mathbb{N}} 2^{-K(y|x,K(x))} \leq 1$ *for fixed* $x \in \mathbb{N}$, *and*
4. $\sum_{y\in\mathbb{N}} 2^{-K(x,y)} \leq 1$ *for fixed* $x \in \mathbb{N}$.

Proof. For $x \in \mathcal{B}^*$, let p_x be a minimal program that computes x. That is, $U(p_x) = x$ and $l(p) = K(x)$. Since $\{p_x : x \in \mathcal{B}^*\}$ is contained in the set of programs of U, it is prefix free. The first inequality then follows straightforwardly from Kraft's inequality. The proofs of the other three inequalities are similar. □

Theorem 3 (input symmetry)**.** *The Kolmogorov complexity of the pairs (x,y) and (y,x) are equal:*

$$K(x,y) = K(y,x)$$

up to an additive constant independent of x and y.

Proof. Suppose p is a program such that $U(p) = (x,y)$ and $l(p) = K(x,y)$. There exists a Turing machine T_i with $i = O(1)$ that simulates U and then, if U outputs a pair, T_i reverses the ordering of the outputted pair.. So $T_i(p) = (y,x)$. Since $U(i,p) = (y,x)$, it follows that $K(y,x) \leq l(i,p) = K(x,y) + O(1)$. The other inequality is similar. □

Much more can be said about Kolmogorov complexity. The definitive resources are [6] and [1].

5 Computability

In this section, we introduce a number of concepts of computability that are necessary for the next section.

Definiton 8. *A partial function $f : \mathbb{N} \to \mathbb{N}$ is **partial recursive** if there exists a Turing machine T such that T computes f. That is, $f(x) = T(x)$, when $f(x)$ is defined and $T(x) = \infty$ otherwise. A total function that is partial recursive is called a **(total) recursive** function.*

Partial recursive functions $f : \mathbb{N}^n \to \mathbb{N}^m$ are defined similarly. We can think of a function $f : \mathbb{N}^2 \to \mathbb{N}^2$ as a function on $\mathbb{Q}^{\geq 0} = \{q = \frac{a}{b} : a,b \in \mathbb{N}, b \neq 0\}$ by interpreting $f(a,b) = (c,d)$ as $f(\frac{a}{b}) = \frac{c}{d}$.

Definiton 9. *A partial function $f : \mathbb{Q}^{\geq 0} \to \mathbb{R}^{\geq 0}$ from the non-negative rationals to the non-negative reals is **computable** if there exists a partial recursive function $\varphi : \mathbb{Q}^{\geq 0} \times \mathbb{N} \to \mathbb{Q}^{\geq 0}$ such that $f(x) = \lim_{t\to\infty} \varphi(x,t)$ for all $x \in \mathbb{Q}^{\geq 0}$.*

We call such a φ an approximator of f. Often we consider the approximator φ and the computable function f as essentially the same object and swap between them as is convenient. Similarly, for each partial recursive function f, there is a Turing machine that computes f and each Turing machine defines a partial recursive function. So we can consider Turing machines and partial recursive functions as essentially the same and interchange them when we like.

Definiton 10. *A partial function $f : \mathbb{Q}^{\geq 0} \to \mathbb{R}^{\geq 0}$ is **lower semi-computable** (resp. **upper semi-computable**) if there exists a partial recursive function $\varphi : \mathbb{Q}^{\geq 0} \times \mathbb{N} \to \mathbb{Q}^{\geq 0}$ such that*

1. *$f(x) = \lim_{t\to\infty} \varphi(x,t)$ for all $x \in \mathbb{Q}^{\geq 0}$, and*
2. *$\varphi(x,t) \leq \varphi(x,t+1)$ (resp. $\varphi(x,t) \geq \varphi(x,t+1)$) for all $x \in \mathbb{Q}^{\geq 0}$ and $t \in \mathbb{N}$.*

A lower semi-computable function is computable with the added assumption that there is an approximator which is non-decreasing in terms of t. In the case of an upper semi-computable function, the approximator is non-increasing.

It is perhaps surprising that there are uncomputable functions. The canonical example is the halting function h:

$$h(i,x) = \begin{cases} 1 & \text{if } T_i(x) \text{ halts,} \\ 0 & \text{otherwise.} \end{cases}$$

See [7, section 1-5.2] for a reference. Another example is Kolmogorov complexity K. See [6, section 3.4] for a proof. However, we have the following positive result.

Proposition 2. *Kolmogorov complexity $K : \mathbb{N} \to \mathbb{N}$ is upper semi-computable.*

Proof. We will construct a Turing machine T that finds better and better estimates of K. In section 3, we described how to enumerate all of the binary strings. Let $b_1, b_2, b_3, ...$ be this enumeration. Using this list, construct T in the following way: firstly, T simulates U with input b_1 for one step, then T simulates U with input b_1 and separately with input b_2, each for two steps and so on, so that at the ith stage, T simulates U on the first i binary strings, each for i steps. This method of computation is called dovetailing.

Let T have input x. Initialise T's estimate P of $K(x)$ as, say, $l(x) + 1000$. (If we had an encoding of ∞ for Turing machines, we would initialise $P = \infty$. Instead, we assume $l(x) + 1000$ is an upper bound for $K(x)$. This depends on our choice of universal Turing machine U, but it seems reasonable that there exists a short string $b \in \mathcal{B}^*$ with $l(b) < 1000$ such that $U(bx) = x$ for all $x \in \mathcal{B}^*$. If we do not have such a b, then simply increase the bound from 1000 until we do. Note that this bound is constant as long as our universal Turing machine U is fixed.) Now, if U halts on some input b_i then T checks whether the output is x. If it is and $l(b_i) < P$, then T updates its estimate $P = l(b_i)$.

Define an approximator φ of K by setting $\varphi(x,t)$ to be the estimate P stored in $T(x)$ after t steps of computation. It is easy to check that φ satisfies the properties in the definition of a upper semi-computable function. □

Corollary 3. *The function $x \mapsto 2^{-K(x)} : \mathbb{N} \to \mathbb{Q}^{\geq 0}$ is lower semi-computable.*

Proof. Modify the Turing machine T in the proof of proposition 2, so that instead of constructing estimates $P = l(b_i)$, it constructs estimates $P = 2^{-l(b_i)}$. □

Importantly, these proofs are constructive. That is, we have actually constructed the Turing machines that approximate $K(x)$ and $2^{-K(x)}$ and it would be possible to find their indices in the list of Turing machines in section 2.

A final concept that will prove to be useful is 'effective enumeration':

Definiton 11. *An **effective enumeration** of a set $S \subset \mathbb{N}$ is a recursive, surjective function $f : \mathbb{N} \to S$. If there is an effective enumeration of S, then we say that S is **effectively enumerable**.*

Often, we wish to talk about effective enumerations of sets other than subsets of $\mathbb{N}$. For example, we might want an effective enumeration of a set of Turing machines or a set of lower semi-computable functions. However, we do not have a definition of recursive functions with codomains other than $\mathbb{Q}^{\geq 0}$. There is an easy (technical) workaround: Since we can encode Turing machines as binary strings and we have a correspondence between $\mathcal{B}^*$ and $\mathbb{N}$, an effective enumeration of a set S of Turing machines is defined as a recursive function onto the subset of $\mathbb{N}$ corresponding to S. Moreover, since there is a one-to-one correspondence between partial recursive functions and Turing machines, this gives a way of defining effective enumerations for sets of partial recursive functions. Finally, each lower semi-computable function f is completely defined by one of its approximators, so we define an effective enumeration of a set S of lower semi-computable functions as an effective enumeration of (recursive) approximators of functions in S.

We often write an effective enumeration with sequence notation. For example, we write $f(1) = a_1, f(2) = a_2, ..., f(n) = a_n, ...$ as the effective enumeration f. Recall in section 3, we showed that $\mathcal{B}^*$ can be effectively enumerated. Additionally, the enumeration of Turing machines at the end of section 2 is effective. Why? Define the effective enumeration f inductively: first effectively enumerate $\mathcal{B}^*$ using $\mathcal{N}$ in section 3. It is a standard result that there exists a Turing machine T that can determine whether a string is an encoding of some Turing machine. Use T to find the first string in the enumeration of $\mathcal{B}^*$ that is an encoding of some Turing machine, say T_1. Define $f(1) = T_1$. Then, find the next string that is an encoding of Turing machine T_2 and define $f(2) = T_2$. Continue this process to define $f(n)$ for arbitrary $n \in \mathbb{N}$. Since there is a correspondence between partial recursive functions and Turing machines, this enumeration of Turing machines gives an effective enumeration of partial recursive functions as well.

6 Universality and the Coding Theorem

In this section, we introduce the concepts of universality and dominance. We present the standard coding theorem, which is of theoretic interest in its own regard but will also be needed to prove lemma 6. Then we develop a modified coding theorem, which will form the central part of the proof of SAI.

Definiton 12. *Given a set of functions $\mathcal{S}$, a function $f \in \mathcal{S}$ is a **universal element** of $\mathcal{S}$ if, for all $g \in \mathcal{S}$, there is some constant c_g such that*

$$f(y) \geq c_g g(y)$$

*for all $y \in \mathbb{N}$. In this case, we say that f is **universal** in $\mathcal{S}$ and **dominates** each $g \in \mathcal{S}$. We call c_g the **dominating constant** of g.*

Usually, we only consider universality when the functions are non-negative and consequently, the constant is usually positive.

Definiton 13. *Define $\mathcal{M}$ to be the set of lower semi-computable functions $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ such that $\sum_{x \in \mathbb{N}} f(x) \leq 1$.*

Lemma 1. *There is an effective enumeration $f_1, f_2, ..., f_n, ...$ of $\mathcal{M}$. Moreover, the function*

$$m(x) = \sum_{n \in \mathbb{N}} 2^{-K(n)} f_n(x)$$

is a universal element of $\mathcal{M}$.

There are in fact infinitely many universal functions of $\mathcal{M}$. We will fix m as the reference universal function of $\mathcal{M}$ (in the same way that we fixed the reference universal Turing machine), since its form will be convenient later.

The proof of lemma 1 is deferred to section 8 but an outline is as follows: The effective enumeration of $\mathcal{M}$ is obtained by modifying the effective enumeration $g_1, g_2, g_3, ...$ of the partial recursive functions given at the end of section 5. We will show that we can modify each g_n in such a way that the result g'_n is in $\mathcal{M}$ and

$g'_n = g_n$ if g_n is in $\mathcal{M}$. Then, since every function in $\mathcal{M}$ is in the enumeration $g_1, g_2, g_3, ...$ already, $g'_1, g'_2, ...$ is an effective enumeration of $\mathcal{M}$. The second part of the lemma, that m is universal in $\mathcal{M}$, then follows straightforwardly.

Theorem 4 (the coding theorem)**.** *For all $x \in \mathbb{N}$,*

$$-\log m(x) = K(x)$$

with equality up to an additive constant independent of x.

To prove the coding theorem, we will need the lemma below whose proof we will defer to section 8.

Lemma 2 (the decoding lemma)**.** *There is a Turing machine T and a prefix code $E : \mathbb{N} \to \mathcal{B}^*$ such that*

1. *T decodes E, in the sense that $T(y) = x$ if $y = E(x)$ for some $x \in \mathbb{N}$, and*
2. *$l(E(x)) \leq -\log m(x) + 3$ for all $x \in \mathbb{N}$.*

Proof of the coding theorem (theorem 4). We will prove $-\log m(x) \leq K(x) + O(1)$ and $-\log m(x) \geq K(x) + O(1)$.

"$\leq$": Let $h : \mathbb{N} \to \mathbb{Q}^{\geq 0}$ be the function defined by $h(x) = 2^{-K(x)}$. We know that h is lower semi-computable by corollary 3. Also, $\sum_{x\in\mathbb{N}} h(x) \leq 1$ by corollary 2.1. Thus, $h \in \mathcal{M}$ and consequently, m dominates h.

"$\geq$": Lemma 2 gives $K_T(x) \leq -\log m(x) + 3$ for all $x \in \mathbb{N}$. Therefore, by the invariance theorem (theorem 2),

$$K(x) \leq K_T(x) + O(1) \leq -\log m(x) + O(1).$$

□

We will now develop a modified version of the coding theorem.

Definiton 14. *Define $\mathcal{M}'$ to be the set of functions $f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^{\geq 0}$ such that:*

1. *for fixed $x \in \mathbb{N}$, the sum $\sum_{y\in\mathbb{N}} f(x, y)$ is bounded by some constant $c \geq 1$ independent of x, y and f.*
2. *$f(x, y)$ is lower semi-computable given $K(x)$. That is, there exists a Turing machine T such that T, with side information $K(x)$, approximates $f(x, y)$ from below:*
   (a) *$\lim_{t\to\infty} T(K(x), (x, y), t) = f(x, y)$ for all $x, y \in \mathbb{N}$, and*
   (b) *$T(K(x), (x, y), t) \leq T(K(x), (x, y), t + 1)$ for all $x, y, t \in \mathbb{N}$.*

The constant c is needed so that the function $g(x, y) = 2^{K(x)-K(x,y)}$ is in $\mathcal{M}'$. Why this is necessary will become apparent in the proof of SAI (theorem 6). The exact value of c will be specified at the end of the proof of lemma 6 and should not concern the reader at the moment.

We will construct a universal element $m'(x, y) = \sum_{n\in\mathbb{N}} 2^{-K(n)} f_n(x, y)$ of $\mathcal{M}'$. Since this is analogous to lemma 1, we leave the details until lemma 3 of section 8.

Theorem 5 (the modified coding theorem)**.** *For all $x, y \in \mathbb{N}$,*

$$-\log m(x, y) = K(y|x, K(x))$$

with equality up to an additive constant independent of x and y.

It follows that the function $(x, y) \mapsto 2^{-K(y|x,K(x))}$ dominates every function in $\mathcal{M}'$. Note that to prove SAI, we only need that $-\log m(x, y) \geq K(y|x, K(x)) + O(1)$. Unfortunately, this turns out to be the hard inequality. The proof of the modified coding theorem is analogous to the proof of the coding theorem, so it will be delayed until section 8.

7 Symmetry of Algorithmic Information

We are now at the point where we can properly state and prove the symmetry of algorithmic information theorem.

Theorem 6. *For all $x, y \in \mathbb{N}$,*

$$K(x) + K(y|x, K(x)) = K(x, y) = K(y, x) = K(y) + K(x|y, K(y)),$$

where each equality is up to an additive constant, independent of x and y.

Proof. Using theorem 3 (input symmetry), it suffices to show $K(x, y) = K(x) + K(y|x, K(x))$. We will show $K(x, y) \leq K(x) + K(y|x, K(x)) + O(1)$ and $K(x, y) \geq K(x) + K(y|x, K(x)) + O(1)$ separately.

"$\leq$": Suppose we have minimal length programs p and q such that $U(p) = x$, $l(p) = K(x)$, $U(q|x, K(x)) = y$ and $l(q) = K(y|x, K(x))$. Construct a Turing machine T_i that takes p and q as input, computes $U(p) = x$ and then computes q on U using side information x and $l(p) = K(x)$. We get $T_i(pq) = (x, y)$ and so $K_{T_i}(x, y) \leq l(p) + l(q) = K(x) + K(y|x, K(x))$. The result then follows by the invariance theorem (theorem 2).

"$\geq$": We want to apply the modified coding theorem (theorem 5) to show that $g(x, y) = 2^{K(x)-K(x,y)}$ is dominated by $h(x, y) = 2^{-K(y|x,K(x))}$. Then, by taking logarithms, we would have

$$K(x) - K(x, y) \leq -K(y|x, K(x)) + O(1)$$

from which the result follows.

To apply the modified coding theorem, we need to show that $g \in \mathcal{M}'$. We do this in lemma 6 below. □

8 Proof of the Modified Coding Theorem and Lemmas in sections 6 and 7

In section 6, there are a number of results given without proof. We will prove them in this section. Lastly, we will prove lemma 6, which states that the function g in the proof of SAI (theorem 6) is an element of $\mathcal{M}'$.

Firstly, we will prove that $m(x) = \sum_{n \in \mathbb{N}} 2^{-K(n)} f_n(x)$ is a universal element of $\mathcal{M}$:

Proof of lemma 1. The outline for this proof is:

1. Construct an effective enumeration $f_1, f_2, ..., f_n, ...$ of $\mathcal{M}$.
2. Use this enumeration to define $m(x) = \sum_{n \in \mathbb{N}} 2^{-K(n)} f_n(x)$.
3. Show that $m \in \mathcal{M}$. Specifically, show that m is lower semi-computable and $\sum_{x \in \mathbb{N}} m(x) \leq 1$.
4. Show that m dominates every function $f \in \mathcal{M}$.

To do part 1, we modify the enumeration of partial recursive functions $g_1, g_2, ...$ given at the end of section 5 to construct an effective enumeration of $\mathcal{M}$. We do this in two stages. In the first stage, we modify $g_1, g_2, ...$ to construct an effective enumeration $g'_1, g'_2, ...$ of lower semi-computable functions. In the second stage, we further modify this enumeration to get an enumeration $f_1, f_2, ...$ so that the sums $\sum_{x \in \mathbb{N}} g'_n(x)$ are all bounded by 1. Then $f_1, f_2, ...$ will be an effective enumeration of functions of $\mathcal{M}$.

Stage 1: Let g_n be some function in the enumeration $g_1, g_2, ...$ The function g_n can be considered as having two arguments x and t by simply defining $g_n(x, t) = g_n((x, t))$. We will modify g_n to construct a two-argument partial recursive function $g'_n(x, t)$ that satisfies the following properties: for all $x, t \in \mathbb{N}$,

1. if $g'_n(x, t) < \infty$ then $g'_n(x, 0), ..., g'_n(x, t - 1) < \infty$, and
2. $g'_n(x, t) \leq g'_n(x, t + 1)$.

To guarantee that the first property is satisfied, dovetail the computation of $g_n(x,0), g_n(x,1), ...$ and assign the computed values to $g'_n(x,0), g'_n(x,1), ...$ in order of computation time. To guarantee that the second property is satisfied, when we assign the computed values of $g_n(x,0), g_n(x,1), ...$ to $g'_n(x,0), g'_n(x,1), ...$, only assign values that satisfy $g'_n(x,t+1) \geq g'_n(x,t)$ and ignore all the other computed values.

If g_n approximates a lower semi-computable function from below, then $g'_n = g_n$. That is to say, we have not modified any of the approximators of lower semi-computable functions. But because of the modifications, every g'_n approximates a lower semi-computable function from below. So $g'_1, g'_2, ...$ is an effective enumeration of (approximators of) all lower semi-computable functions.

Stage 2: Again, consider an arbitrary function g'_n in the enumeration $g'_1, g'_2, ...$ Construct a Turing machine T that contains an array P and does the following:

1. Initialise T by setting $P(x) := 0$ for all $x \in \mathbb{N}$ and set $t := 0$.
2. The machine T compute $g'_n(0,t), g'_n(1,t), ..., g'_n(t,t)$. (If any of the $g'_n(x,t)$ is undefined, for $0 \leq x \leq t$, then T will continue computing $g'_n(x,t)$ indefinitely, and P will remain unchanged.)
3. If $g'_n(0,t) + g'_n(1,t) + ... + g'_n(t,t) \leq 1$ then T sets $P(x) := g'_n(x,t)$ for all $x = 0, ..., t$. If $g'_n(0,t) + g'_n(1,t) + ... + g'_n(t,t) > 1$, then T terminates.
4. Set $t := t + 1$ and go to step 2.

Note that the array P can be stored on the working tape of T using a finite amount of memory, since at any point only finitely many values $P(x)$ are non-zero.

Define f_n in the following way: To calculate $f_n(x,t)$, run T for t time steps. Then observe the current state of the array P stored in T. Define $f_n(x,t) = P(x)$. By construction, for all $t \in \mathbb{N}$,

$$\sum_{x \in \mathbb{N}} f_n(x,t) \leq 1.$$

If g_n approximates some function $h \in \mathcal{M}$ from below, then f_n also approximates h from below. Also, all of the approximators of functions in $\mathcal{M}$ are in the original enumeration $g_1, g_2, ...$ of partial recursive functions. The resulting effective enumeration $f_1, f_2, ...$ must then still contain approximators for all functions in $\mathcal{M}$. By construction $f_1, f_2, ...$ only contains approximators of functions in $\mathcal{M}$. Thus, it is an effective enumeration of $\mathcal{M}$. This proves part 1. of the proof outline.

Part 2. requires no explanation. We prove part 3. now. The function m is lower semi-computable. Why? Suppose φ_n is an approximator of f_n from below and ψ is an approximator of $n \mapsto 2^{-K(n)}$ from below. (The existence of φ_n and ψ are guaranteed by the fact that f_n and $n \mapsto 2^{-K(n)}$ are lower semi-computable.) Then

$$\varphi(x,t) = \sum_{n=1}^{t} \psi(n,t)\varphi_n(x,t)$$

is an approximator of m from below.

To prove the second half of part 3., expanding the definition of m gives

$$\sum_{x \in \mathbb{N}} m(x) = \sum_{x \in \mathbb{N}} \sum_{n \in \mathbb{N}} 2^{-K(n)} f_n(x).$$

Since all of the terms in the sum are non-negative, we can interchange the order of the sums and obtain

$$\sum_{x \in \mathbb{N}} \sum_{n \in \mathbb{N}} 2^{-K(n)} f_n(x) = \sum_{n \in \mathbb{N}} 2^{-K(n)} \sum_{x \in \mathbb{N}} f_n(x) \leq \sum_{n \in \mathbb{N}} 2^{-K(n)}.$$

Corollary 2.1 states that $\sum_{n \in \mathbb{N}} 2^{-K(n)} \leq 1$, which gives us the required result: $\sum_{x \in \mathbb{N}} m(x) \leq 1$.

The proof of part 4. is straightforward. Suppose $f \in \mathcal{M}$. Then $f = f_j$ for some $j \in \mathbb{N}$ in our enumeration of $\mathcal{M}$. By construction of m, we get $m(x) \geq 2^{-K(j)} f_j(x)$ for all $x \in \mathbb{N}$. The dominating constant is $2^{-K(j)}$, which obviously does not depend on x, as required. □

Proof of the decoding lemma (lemma 2). We will construct a code E, in a similar way to the construction of a Shannon-Fano code. (See [2, section 5.9] for a reference on Shannon-Fano codes.) Let φ be an approximator of m from below. (The existence of φ is guaranteed since m is lower semi-computable.) Define a new recursive function $\psi : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ by:

$$\psi(x,t) = \begin{cases} 2^{-k} & \text{if } 2^{-k} \leq \varphi(x,t) < 2^{-k+1} \text{ and } \varphi(x,j) < 2^{-k} \text{ for all } j < t, \\ 0 & \text{otherwise.} \end{cases}$$

Consider the function $t \mapsto \psi(x,t)$ for fixed x. The range of this function is a subset of $\{2^{-k} : k \in \mathbb{N}, 2^{-k} \leq m(x)\} \cup \{0\}$. Moreover, each non-zero value 2^{-k} is achieved at most once by $t \mapsto \psi(x,t)$. Thus,

$$\sum_{x \in \mathbb{N}, t \in \mathbb{N}} \psi(x,t) = \sum_{x \in \mathbb{N}} \sum_{t \in \mathbb{N}} \psi(x,t) \leq \sum_{x \in \mathbb{N}} \sum_{k \in \mathbb{N} : 2^{-k} \leq m(x)} 2^{-k} \leq \sum_{x \in \mathbb{N}} \sum_{k \in \mathbb{N}} 2^{-k} m(x) = \sum_{x \in \mathbb{N}} 2m(x) \leq 2.$$

Importantly, we have a bound on the sum $\sum_{x \in \mathbb{N}, t \in \mathbb{N}} \psi(x,t)$. We use this to construct the decoding Turing machine T in the following way. Chop off consecutive, adjacent, disjoint half-open intervals $I_{x,t}$ of length $\psi(x,t)/2$ starting from the left side of $[0,1)$. (Technically, we need to find an ordering of $\{(x,t) : x,t \in \mathbb{N}\}$ so that we have an ordering of the intervals $I_{x,t}$. To do this, dovetail the computations of all the $\psi(x,t)$ and order the tuples (x,t) by computation time.)

Recall the definition of cylinder sets Γ_p for $p \in \mathcal{B}^*$ from the proof of Kraft's inequality (theorem 1). If Γ_p is the largest cylinder set of $I_{x,t}$ then define $T(p) = x$. (If there are two cylinder sets of $I_{x,t}$ that are equally as large, choose the leftmost one.) Otherwise, define $T(p) = \infty$.

Now we will define E. (Note that there is no requirement that E is recursive.) Because the length of $I_{x,t}$ is of the form 2^{-k} for some $k \in \mathbb{N}$, the interval $I_{x,t}$ contains a cylinder set of length 2^{-k-1}. By construction of ψ, for each $x \in \mathbb{N}$, there is some $t \in \mathbb{N}$ such that $\psi(x,t) > m(x)/2$. Thus, for each $x \in \mathbb{N}$, there is an interval $I_{x,t}$ of length greater than $m(x)/4$ and consequently, there is a cylinder set Γ_p (contained in $I_{x,t}$) of length greater than $m(x)/8$. Then define the code $E : \mathbb{N} \to \mathcal{B}^*$ by $E(x) = p$.

By definition of the cylinder set, the length of Γ_p is $2^{-l(p)}$. Thus,

$$2^{-l(E(x))} > m(x)/8.$$

This gives the result $l(E(x)) < -\log m(x) + 3$, for all $x \in \mathbb{N}$, as required. □

We have proved all the results needed to establish the (unmodified) coding theorem. Now, we will prove the modified coding theorem. First, we need to show to construct our universal element m' of $\mathcal{M}'$:

Lemma 3. *There is an effective enumeration $f'_1, f'_2, ..., f'_n, ...$ of $\mathcal{M}'$. Moreover, the function*

$$m'(x,y) = \sum_{n \in \mathbb{N}} 2^{-K(n)} f'_n(x,y)$$

is a universal element of $\mathcal{M}'$.

Proof. The proof is very similar to the proof that m is a universal element of $\mathcal{M}$ (see lemma 1). We construct an effective enumeration $f'_1, f'_2, ...$ of $\mathcal{M}'$ in the same way, with the following exceptions:

1. Given a function g_n of the effective enumeration $g_1, g_2, ...$ of partial recursive functions, we consider g_n as having 3 arguments x, y and t.
2. We modify g_n to construct a three-argument partial recursive function $g'_n(x,y,t)$ that satisfies the following properties: for all $x, y, t \in \mathbb{N}$
   (a) if $g'_n(x,y,t) < \infty$ then $g'_n(x,y,0), ..., g'_n(x,y,t-1) < \infty$, and
   (b) $g'_n(x,y,t) \leq g'_n(x,y,t+1)$.

The construction of g'_n is by dovetailing $g_n(x,y,0), g_n(x,y,1), ...$ and selecting computed values in order of computation time as in lemma 1.

3. The Turing machine T in stage 2 does the following:
   (a) Initialise T by setting $P(x,y) := 0$ for all $x, y \in \mathbb{N}$.
   (b) Then, T dovetails the following computation for all $x \in \mathbb{N}$:
      i. Set $t := 0$.
      ii. Compute $g'_n(x,0,t), g'_n(x,1,t), ..., g'_n(x,t,t)$.
      iii. If $g'_n(x,0,t) + g'_n(x,1,t) + ... + g'_n(x,t,t) \leq c$, then set $P(x,y) := g'_n(x,y,t)$ for all $y = 0, ..., t$. If $g'_n(x,0,t) + g'_n(x,1,t) + ... + g'_n(x,t,t) > c$, then the computation terminates. (Recall, c is the constant bound in the definition of $\mathcal{M}'$ (see definition 14).)
      iv. Set $t := t + 1$ and go to step ii.

We can prove $m' \in \mathcal{M}'$ in exactly the same way as we did for $m \in \mathcal{M}$. Similarly, the proof that m' dominates each function in $\mathcal{M}'$ follows analogously to the case for m. □

To prove the modified coding theorem, we need a result analogous to the decoding lemma (lemma 2):

Lemma 4 (the modified decoding lemma)**.** *There is a Turing machine T and prefix codes $E_x : \mathbb{N} \to \mathcal{B}^*$ for each $x \in \mathbb{N}$ such that*

1. *Given side information x and $K(x)$, the machine T decodes E, in the sense that, for each $x \in \mathbb{N}$,*

$$T((x, K(x)), z) = y \text{ if } z = E_x(y) \text{ for some } y \in \mathbb{N},$$

and

2. $l(E_x(y)) \leq -\log m(x,y) + \log c + 3$ *for all* $y \in \mathbb{N}$.

Importantly, while the prefix codes E_x can vary with x, the decoding Turing machine T does not.

Proof. Once again, this proof is very similar to the unmodified case (lemma 2). It can basically be copied word for word, with a couple key changes. Let φ be an approximator of m from below given $K(x)$ as side information. Define a new recursive function $\psi : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ by:

$$\psi(x,y,t) = \begin{cases} 2^{-k} & \text{if } 2^{-k} \leq \varphi(x,y,t) < 2^{-k+1} \text{ and } \varphi(x,y,j) < 2^{-k} \text{ for all } j < t, \\ 0 & \text{otherwise.} \end{cases}$$

Fix $x \in \mathbb{N}$. In exactly the same way as in the proof of the unmodified case, we get the bound

$$\sum_{y \in \mathbb{N}, t \in \mathbb{N}} \psi(x,y,t) \leq 2c.$$

We construct the decoding Turing machine T as before, with some minor changes. Chop off consecutive, adjacent, disjoint half-open intervals $I_{x,y,t}$ of length $\psi(x,y,t)/(2c)$ starting from the left side of $[0,1)$. If Γ_p is the largest cylinder set of $I_{x,y,t}$ then define $T((x, K(x)), p) = y$. (We need to give $K(x)$ as side information, otherwise T would not be able to compute ψ. And T needs x as side information, otherwise it would not be able to find the intervals $I_{x,y,t}$.) Otherwise, define $T(p) = \infty$.

Again define E_x analogously to the unmodified case in lemma 2. For each $y \in \mathbb{N}$, there is an interval $I_{x,y,t}$ of length greater than $m(x,y)/(4c)$ and consequently, there is a cylinder set Γ_p (contained in $I_{x,y,t}$) of length greater than $m(x,y)/(8c)$. Then, define the code $E_x : \mathbb{N} \to \mathcal{B}^*$ by $E_x(y) = p$.

The length of Γ_p is $2^{-l(p)}$. Thus,

$$2^{-l(E_x(y))} > m(x,y)/(8c).$$

This gives the result $l(E_x(y)) < -\log m(x,y) + \log c + 3$, for all $x, y \in \mathbb{N}$, as required. □

Before we prove the modified coding theorem, we need one more lemma:

Lemma 5. *The function* $h(x,y) = 2^{-K(y|x,K(x))}$ *is in* $\mathcal{M}'$.

Proof. We need to show that

1. h is lower semi-computable given $K(x)$, and
2. for fixed $x \in \mathbb{N}$, the sum $\sum_{y\in\mathbb{N}} h(x,y)$ is no greater than the constant c.

(Recall c is the constant specified in the definition of $\mathcal{M}'$ (definition 14).) We proved $\sum_{y\in\mathbb{N}} h(x,y) \leq 1$ for fixed $x \in \mathbb{N}$ in corollary 2.3. Since $c \geq 1$ by definition, we have shown 2.

The proof of 1. is analogous to the proof that $x \mapsto 2^{-K(x)}$ is lower semi-computable (see corollary 3): Suppose $b_1, b_2, ...$ is the enumeration of binary strings given in section 3. Construct a Turing machine T that receives input (x,y) and side information $K(x)$ in the following way. First T dovetails the computation of U on inputs $b_1, b_2, b_3, ...$ with side information $K(x)$ and x. That is, T dovetails the computations of $U((x,K(x)),b_1), U((x,K(x),b_2), ...$ If the simulation of $U((x,K(x)),b_n)$ halts, T checks that the output is equal to y. If it is, then $2^{-l(b_n)}$ is a lower bound on $h(x,y)$. Using T, define an approximator φ to $h(x,y)$ in the obvious way (see corollary 3) such that φ satisfies the properties in the definition of lower semi-computability. □

We are now ready to prove the modified coding theorem.

Proof of the modified coding theorem (theorem 5). This proof is very similar to the proof of the (unmodified) coding theorem (theorem 4). We will prove $-\log m(x,y) \leq K(y|x,K(x)) + O(1)$ and $-\log m(x,y) \geq K(y|x,K(x)) + O(1)$.

"$\leq$": Let $h : \mathbb{N}\times\mathbb{N} \to \mathbb{Q}^{\geq 0}$ be the function defined by $h(x,y) = 2^{-K(y|x,K(x))}$. We know that $h \in \mathcal{M}'$ by lemma 5 and consequently, m' dominates h.

"$\geq$": Lemma 4 gives $K_T(y|x,K(x)) \leq -\log m(x) + \log c + 3$ for all $x, y \in \mathbb{N}$. Therefore, by a simple variant of the invariance theorem (theorem 2),

$$K(y|x,K(x)) \leq K_T(y|x,K(x)) + O(1) \leq -\log m(x,y) + O(1).$$

□

We have now proved all the results in section 6. Finally, we prove the lemma used at the end of the proof of SAI (theorem 6).

Lemma 6. *The function* $g(x,y) = 2^{K(x)-K(x,y)}$ *is in* $\mathcal{M}'$.

Proof. To show g is lower semi-computable, construct a Turing machine T' similar to the machine T in the proof of lemma 5. Suppose $b_1, b_2, ...$ is the enumeration of binary strings given in section 3. The machine T' receives input (x,y) and side information $K(x)$. Then T' dovetails the computations of $U(b_1), U(b_2), ...$ Since each b_n with $U(b_n) = (x,y)$ is an upper bound of $K(x,y)$, the machine T' constructs lower bounds $2^{K(x)-l(b_n)}$ of $g(x,y)$. The rest of the construction of T' and approximator φ' is the same as in the proof of lemma 5.

Now we will show $\sum_{y\in\mathbb{N}} g(x,y)$ is uniformly bounded for fixed $x \in \mathbb{N}$. This follows from Eq.(11f) in [5]. In Eq.(11f), define $f(z) = x$ if $z = (x,y)$ and undefined otherwise. We also present an elementary proof: By definition,

$$\sum_{y\in\mathbb{N}} 2^{-K(x,y)} = \sum_{y\in\mathbb{N}} \max_{p:U(p)=(x,y)} 2^{-l(p)} \leq \sum_{y\in\mathbb{N}} \sum_{p:U(p)=(x,y)} 2^{-l(p)}.$$

Notice that the last summation above is over all programs that output (x,y) for any y. Construct a Turing machine T_i that does the following: given input p, first T_i simulates $U(p)$. If U outputs a tuple (x,y) then T_i outputs the first element x. So if $U(p) = (x,y)$ then $U(i,p) = x$. For every $y \in \mathbb{N}$, we have modified

every program p that outputs (x, y) into a program (i, p) that outputs x. The difference between the lengths of p and $(i, p) = \mathcal{E}(i)p$ is $l(\mathcal{E}(i)) = 2l(i) + 1$. Thus,

$$\sum_{y\in\mathbb{N}} \sum_{p:U(p)=(x,y)} 2^{-l(p)} = 2^{2l(i)+1} \sum_{y\in\mathbb{N}} \sum_{p:U(p)=(x,y)} 2^{-l(i,p)} \leq 2^{2l(i)+1} \sum_{q:U(q)=x} 2^{-l(q)}.$$

We will show that the function $f(x) = \sum_{q:U(q)=x} 2^{-l(q)}$ is in $\mathcal{M}$. Then, by the coding theorem (theorem 4), f is dominated by $x \mapsto 2^{-K(x)}$. Let c_f be the dominating constant. This gives us the result:

$$\sum_{y\in\mathbb{N}} 2^{K(x)-K(x,y)} = 2^{K(x)} \sum_{y\in\mathbb{N}} 2^{-K(x,y)} \leq 2^{K(x)} \left(2^{2l(i)+1} \sum_{q:U(q)=x} 2^{-l(q)} \right) \leq 2^{K(x)} \left(2^{2l(i)+1} c_f 2^{-K(x)} \right) = 2^{2l(i)+1} c_f.$$

It remains to show that $f \in \mathcal{M}$. The bound on the sum

$$\sum_{x\in\mathbb{N}} f(x) = \sum_{x\in\mathbb{N}} \sum_{q:U(q)=x} 2^{-l(q)} \leq \sum_{p \text{ a program of } U} 2^{-l(p)} \leq 1,$$

follows from Kraft's inequality (theorem 1). Also, f is lower semi-computable. Why? We prove this in the standard way, by constructing a Turing machine T that obtains increasingly more accurate lower bounds on f. The machine T takes x as input, then dovetails the computation of $U(b_1), U(b_2), ...$, where $b_1, b_2, ...$ is the effective enumeration of binary strings given in section 3. Initialise the lower bound $P = 0$. Then, if some $U(b_n)$ halts and outputs x, the machine T calculates a new lower bound $P' = P + 2^{-l(b_n)}$. Using T, we can construct an approximator φ of f from below in the obvious way. (See, for an example, the proof of proposition 2.) □

We have now given all of the proofs not provided in sections 6 and 7. Before we finish, we make some important observations about the proof above. Firstly, if T' was not allowed to receive side information $K(x)$, T' would not be able to calculate the bounds $2^{K(x)-l(b_n)}$. (In fact, $2^{K(x)-l(b_n)}$ is not even lower semi-computable, let alone computable.) This is why we allow functions in $\mathcal{M}'$ to only be lower semi-computable if given $K(x)$. Consequently, this is why the $K(x)$ term must be included in $K(y|x, K(x))$ in the statement of SAI (theorem 6).

Secondly, the constant $2^{2l(i)+1}c_f$ above does not depend on x or y. Since the function $f(x) = \sum_{q:U(q)=x} 2^{-l(q)}$ is in $\mathcal{M}$, there exists some index j of the effective enumeration $f_1, f_2, f_3, ...$ of $\mathcal{M}$ such that $f = f_j$. Then the dominating constant c_f is $2^{-K(j)}$. We can now specify the constant $c = \max\{2^{2l(i)-K(j)+1}, 1\}$ in the definition of $\mathcal{M}'$ (definition 14). Importantly, this definition of c allows $g(x, y) = 2^{K(x)-K(x,y)}$ to be in $\mathcal{M}'$, which we needed for the proof of SAI (theorem 6).

References

[1] C. S. Calude, *Information and Randomness: An Algorithmic Perspective*, 2nd ed., Springer, Berlin, 2002.

[2] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, John Wiley & Sons, New York, 1991.

[3] P. Gács, *On the symmetry of algorithmic information*, (translated) Soviet Math. Dokl., **15** (1974), 1477-1480.

[4] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., Pearson, New Jersey, 2006.

[5] M. Hutter, *On universal prediction and bayesian confirmation*, Theoretical Computer Science, **384** (2007), 33-48 (available online at `http://www.hutter1.net/ai/uspx.pdf`).

[6] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications* 3rd ed., Springer, New York, 2008.

[7] Z. Manna, *Mathematical Theory of Computation*, Dover Publications, New York, 2003.