Kolmogorov Incompressibility Method in Formal Proofs A Critical Survey

Vasileios Megalooikonomou*
Computer Science Electrical Engineering Department
University of Maryland Baltimore County

January 22, 1997

Abstract

We compare the incompressibility method of Kolmogorov complexity that is used in formal proofs of mathematical and computational results with more traditional methods such as proofs by counting, proofs by probabilistic arguments and proofs by pumping lemmas for formal languages. We consider applications of Kolmogorov complexity in several different areas such as lower bounds, average case analysis of algorithms, formal language theory, and random graphs. We argue that the Kolmogorov complexity proofs are more intuitive, elegant and less lengthy than the other arguments. Furthermore, the former proofs are easier to construct and to understand because all of them have the same structure and they all use a similar argument based on the fact that "almost all" strings are not compressible at all.

Keywords. Computational complexity, Kolmogorov complexity, randomness, compression, k-head finite automata, Turing machines, simulation, lower bounds, average case analysis.

1 Introduction

R.J. Solomonoff (1964), A.N. Kolmogorov (1965), and G.J. Chaitin (1969) (in chronological order) invented an excellent theory of information content of strings, now known as *Kolmogorov complexity*. This theory is useful in a variety of cases from sorting algorithms to combinatorics and theory of computation and from inductive reasoning to entropy and dissipationless computing.

*Author's address: Department of Computer Science Electrical Engineering, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250; megaloik@cs.umbc.edu. Phone: (410) 455-2667; fax: (410) 455-3969

Intuitively, the information content of a finite string is the size (literally, number of bits) of the shortest program that, starting with a blank input, prints the string and terminates. This definition can be extended to infinite strings if the program produces element after element forever. Thus, 1^n (a string of n 1's) contains little information because the following program outputs it:

$$\text{for } i := 1 \text{ to } n \text{ do print('1')}.$$

This program is of size $O(\log n)$ because it contains a description of the number n plus some constant additive terms. An example of an infinite sequence that contains $O(1)$ information is the number $\pi = 3.1415\ldots$. The digits of this number can be produced for ever by a fixed short program.

In this paper we consider applications of Kolmogorov complexity in several different areas such as lower bounds, average case analysis of algorithms, formal language theory, and random graphs. We compare the Kolmogorov complexity arguments that are used in formal proofs of mathematical and computational results with more traditional arguments such as counting arguments and probabilistic arguments. In the case of formal languages we compare the Kolmogorov arguments with the pumping lemmas for formal languages. In order to compare the different approaches we consider the following criteria: readability, length, intuitiveness, elegance, and simplicity in constructing the proof.

Although Kolmogorov complexity contains rich and deep mathematics, the amount of this mathematics one needs to know to apply the notion in the areas that we consider here is little. The necessary background of the mathematical theory of Kolmogorov complexity is treated in Section 2. At the end of Section 2 we illustrate, with a simple example, the use of Kolmogorov complexity in formal mathematical proofs. The following four sections give more elaborate examples and compare the use of Kolmogorov complexity arguments with the more traditional arguments that we mentioned before.

In Section 3 we present the Kolmogorov complexity proofs of two results on lower bounds and we justify the power and elegance of Kolmogorov complexity arguments over the counting arguments. In Section 4 we compare the average case analysis of Heapsort using three approaches, the probabilistic argument, the counting argument, and the Kolmogorov complexity argument. The Kolmogorov complexity replacements of the pumping lemmas for regular expressions and for deterministic context free languages together with selected examples of applications of these lemmas are presented in Section 5. We present and compare the different solutions of several problems (exercises from [3]). Finally, we present an application of Kolmogorov complexity to random graphs. In particular, we present and compare the different proofs of the fact that a certain property holds for almost every tree. We show that this kind of property can be proved very easily and elegantly using Kolmogorov incompressibility arguments. We also point out some further research in this area.

2 Background

In this section we provide an introduction to the most basic type of Kolmogorov complexity. We cover only the essential ideas and useful facts to be used in the applications we present later. The mathematical theory of Kolmogorov complexity extends far beyond our presentation here (see [17] and [9] for a more complete treatment).

We are interested in defining the complexity of a concrete individual finite string of zeros and ones. Unless otherwise specified all strings will be binary and of finite length. All logarithms in this paper are base 2. If x is a string, then $|x|$ denotes the *length* of x. We identify the xth finite binary string in the canonical order with the natural number x, according to the correspondence:

$$(\epsilon, 0), (0, 1), (1, 2), (00, 3), (01, 4), (10, 5), (11, 6), (000, 7) \ldots$$

The canonical order lists words in order of size, with words of the same size in "numerical" order. If A is a set, then $d(A)$ is the *cardinality* of A. Hence, with $A = \{1, 2, \ldots, n\}$ we have $|d(A)|$ to be about $\log n$.

The description of a string depends on two things, the decoding method (the machine that interprets the description) and the outside information available (input to the machine). We restrict the decoders to be Turing machines. Let T be a Turing machine. Then, p is a description of x (i.e. a program) if T outputs x when given as input the program p and some extra information y to help to generate x ($T(p, y) = x$).

Intuitively, we want to call a string simple if it can be described in a few words, like "the string of a million ones". A string is considered complex if it cannot be so easily described. Such a string does not follow any rule and we can think of it as "random" in some sense. Here comes the notion of descriptional complexity of a string.

Definition 1 *The descriptional complexity K_T of a string x, relative to a Turing machine T and a binary string y, is defined by:*

$$K_T(x|y) = \min\{|p| : p \in \{0,1\}^* \text{ and } T(p, y) = x\}$$

The complexity measure defined above is useful and makes sense only if the complexity of a string does not depend on the choice of machine T. Therefore the following theorem is vital.

Theorem 1 Invariance Theorem (Solomonoff [1964], Kolmogorov [1965], Chaitin [1969]). *There exists a (optimal) Turing machine U, such that, for any other Turing machine T, there is a constant c_T such that for all strings $x, y, K_U(x|y) \leq K_T(x|y) + c_T$.*

In other words, the Universal machine U that satisfies the Invariance Theorem is *optimal* in the sense that K_U minorizes each K_T up to a fixed additive constant which depends on U and

T. So the Universal description method does not necessarily give the shortest description in each case, but no other description can improve on it by more than a fixed constant. Moreover, for each pair of Universal Turing machines U and U', satisfying the Invariant Theorem, the complexities coincide up to an additive constant (depending only on U and U'), for all strings x, y:

$$|K_U(x|y) - K_{U'}(x|y)| \leq c_{U,U'}$$

The Kolmogorov complexities according to U and U' are equal up to a fixed constant so we can fix a Universal Turing machine U as the reference Turing machine.

Definition 2 *The canonical conditional Kolmogorov complexity $K(x|y)$ of x under condition y is equal to $K_U(x|y)$, for the fixed Universal Turing Machine U.*

So, now, the Kolmogorov complexity of a string does not depend on the choice of encoding method and is well-defined.

Definition 3 *The unconditional Kolmogorov complexity of x is defined as $K(x) = K(x|\epsilon)$, where ϵ is the empty string ($|\epsilon| = 0$).*

Definition 4 *A binary string x is incompressible if $K(x) \geq |x|$.*

Martin-Löf has shown that incompressible strings pass all effective statistical tests for randomness, so we will also call incompressible strings, *random* strings. However, not all traditionally random strings are Kolmogorov random. π's digits are random according to traditional randomness tests, but quite regular in terms of algorithmic randomness, as we discussed before. A natural question to ask is: how many strings are "random" (incompressible)? It turns out that almost all strings of a given length n are random. We will prove this fact but before we will prove the following:

Fact 1 *For each n, there exists a random string of length n.*

Proof. There are 2^n binary strings of length n, but only $2^n - 1$ possible shorter descriptions. Thus, for all n, there is a binary string x of length n such that $K(x) \geq n$. ■

Fact 2 *Almost all strings of a given length n are random.*

Proof. From the previous argument we know that there is at least one x of length n that cannot be compressed to length $< n$. Using the same argument we can show that at least $\frac{1}{2}$ of all strings of length n cannot be compressed to length $< n - 1$; at least $\frac{3}{4}$th of all strings of length n cannot be compressed to length $< n - 2$, and so on. ■

Generally, let $g(n)$ be an integer function.

Definition 5 *A binary string x of length n is called g-incompressible if $K(x) \geq n - g(n)$.*

There are 2^n binary strings of length n, and only $2^{n-g(n)} - 1$ possible descriptions shorter than $n - g(n)$. Thus, the ratio between the number of strings of length n with $K(x) < n - g(n)$ and the total number of strings of length n is at most $2^{-g(n)}$, a vanishing fraction when $g(n)$ increases unboundedly with n. In general we loosely call a finite string x of length n *random* if $K(x) \geq n - O(\log n)$. Intuitively, incompressibility (*randomness*) implies the absence of regularities, since regularities can be used to compress descriptions.

Fact 3 *All previous facts are true relative to a given binary string y.*

For example Fact 1 can be rewritten relative to string y as:

Fact 4 *For all finite strings y and all n there is an x of length n such that $K(x|y) \geq |x|$.*

Now, we present the following simple but extremely useful *Incompressibility Theorem.*

Theorem 2 Incompressibility Theorem. *Let c be a positive integer. For each fixed y, every finite set A of cardinality m has at least $m(1 - 2^{-c}) + 1$ elements x with $K(x|y) \geq \log m - c$.*

Proof. The number of programs of length less than $\log m - c$ is $\sum_{i=0}^{\log m - c - 1} 2^i = 2^{\log m - c} - 1 = m2^{-c} - 1$. Hence, there are at least $m - m2^{-c} + 1$ elements in A which have no program of length less than $\log m - c$. ■

The deeper reason for this fact is that since there are only few short programs, there can be only few objects of low complexity.

2.1 Self-delimiting descriptions

A string x in $\{0,1\}^*$ with $|x| = n$ can be described by a piece of text containing several formal parameters (i.e. a formal parametrized procedure in an algorithmic language which requires $O(1)$ bits), followed by an ordered list of the actual values for the parameters. To distinguish one from the other, we encode them as self-delimiting strings.

The *self-delimiting* version of a string $y \in \{0,1\}^*$ is the string $y' = \overline{|y|}y$ which is the length of y in binary followed by y in binary. Notice that if we know the length of y and the start of its literal representation we also know where it ends. For a string $z \in \{0,1\}^*$, the string $\overline{z}$ is obtained by inserting a "0" in between each pair of adjacent symbols in z, and adding a "1" at the end. We insert these extra symbols in order to be able to know where the start of the literal representation of y is. That is, $\overline{11111} = 1010101011$. As an example, the self-delimiting

version of '01011' is '100101011' (note that the length of '01011', which is 5, is identified with the 5th binary string (10) in the canonical order as we defined before). Thus, the self-delimiting version of a binary string w requires $|w| + 2\log|w|$ bits and if w is a positive integer n we can substitute "log n" for $|w|$ and have $\log n + 2\log\log n$ bits used for the self-delimiting version.

2.2 An example of using Kolmogorov complexity in formal proofs

Let us take a very simple example from formal language theory just to understand the use of Kolmogorov complexity arguments in formal proofs. We are going to prove that the language $\{0^k1^k : k \geq 1\}$ is not regular (exercise 6.2 in [8]). The proof will be by contradiction such as all the proofs that use Kolmogorov complexity. Suppose that the language is regular. Choose k such that $K(k) \geq \log k$ (i.e. k is Kolmogorov random). Then, the state q of the particular accepting finite automaton (FA) after processing 0^k together with the FA is, up to a constant, a description of k. This holds because by running the FA starting from q on a string containing 1s only the FA accepts only after exactly k consecutive 1s. So, there exist a constant c that depends on the FA such that $\log k < c$, which is a contradiction because the FA should accept the language $\{0^k1^k : k \geq 1\}$ for every k.

3 Proving Lower Bounds

In a traditional lower bound proof by counting, *all inputs* (or all strings of a certain length) are involved, and one shows that the lower bound has to hold for *some* of these inputs (called "typical" inputs). The fact that it is hard to construct such a typical input complicates these proofs because it forces these proofs to consider all the inputs. However, in a Kolmogorov complexity proof, a Kolmogorov random string is considered "typical" input because almost all strings are Kolmogorov random.

A Kolmogorov complexity proof for lower bounds is based on the fact that there are incompressible (or Kolmogorov random) strings. So, we choose a random string that *exists* although it cannot be exhibited or proved to be random or "typical". We use that *fixed, single*, "typical" string to show that if the lower bound does not hold, then this string can be compressed, thus reaching a contradiction. Here, we deal only with one fixed string and this makes the proof easier and more elegant. Also, the lower bounds obtained by Kolmogorov complexity are often stronger than those obtained by its counting counterpart because they hold for "almost all strings" (because almost all strings are random). These lower bounds also imply directly the lower bounds for nondeterministic or probabilistic versions of the machine that we consider.

Some Kolmogorov complexity proofs for lower bounds are based on a method of deleting blocks of bits from a Kolmogorov random string and give an efficient description of the produced string which has "holes" in it (M. Li, W. Maass, P.M.B. Vitányi [15]). The complete string can be reconstructed using an additional description. In this case the contradiction is reached

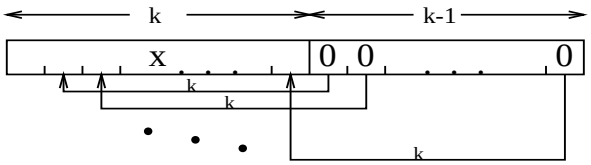

Figure 1: The bit of string x at the head of the arrow can be determined when the head of the FA points at the corresponding 0 bit.

by showing that these two descriptions together have smaller length than the incompressible string they describe. A use of this argument in a specific example is presented in Section 6.

In this section we present two examples of using Kolmogorov complexity to prove lower bounds. The first is a lower bound on the number of states that are needed to convert an NFA to a DFA. The second is a lower bound on the number of one-way heads that are necessary for a DFA to do string matching.

3.1 Convert a NFA to a DFA

Fact 5 *It requires $\Omega(2^n)$ states in the worst case to convert a nondeterministic finite automaton (NFA) with n states to a deterministic finite automaton (DFA).*

Proof.[using Kolmogorov complexity][M. Li and P. M. B. Vitányi, [15]] Consider the language $L_k = \{x |$ the kth bit of x from the right is $1\}$. L_k can be accepted by a NFA with $k+1$ states. Suppose that L_k is accepted by the DFA M which has only $o(2^k)$ states. Let x be a string of length k such that the conditional Kolmogorov complexity of x is $K(x|k, M) \geq |x| = k$, where M is its own description. From Fact 4 we know that such a string exists. Give x as input to DFA M. Record the current state of M after M reads the last bit of x. Now, using the recorded state as the starting state and having the description of M we will reconstruct x. Starting from the recorded state feed M with input consisting of $k-1$ 0's, (actually any string of 0's and 1's with length at least $k-1$ suffices). Notice that this is the same as feeding M with $x0^k$ starting at the start state of M. Notice also that M after reading $x0^{i-1}$ for $2 \leq i \leq k$ accepts it iff the ith bit of x (from the left) is 1. This is easy to see if you recall that x is of length k and M at every step accepts the string which has been read so far if the kth bit from the right is 1. The first bit can be reproduced by considering the recorded state. If that is an accepting state then the first bit of x is 1 otherwise it is 0. Thus, in this way, we can reproduce the string x one bit at a time starting from the first bit (see Figure 1). The information that is needed in order to do that (under the condition that k and M are known) is $k - f(k)$ bits with $f(k)$ unbounded, since we need to record only one state and M has only $o(2^k)$ states. So for large enough k we have $K(x|k, M) < |x|$, which is a contradiction. ■

3.2 String matching: How many one-way heads are necessary for a DFA?

Applying Kolmogorov complexity, Li and Yesha [10] showed that 2-head one-way DFA (or even a Turing machine with two one-way input heads and $o(n)$ storage space) cannot perform string matching (i.e. accept the language $SUBSTR$ =$\{\$x\#y\$|x$ is a substring of y, where $x, y \in \{0,1\}^*$ $\}$). Later, Geréb-Graus and Li [2] showed that this cannot be done even with three heads. Recently, Jiang and Li [4] proved that this cannot be done for any k and settled the conjecture raised by Galil and Seiferas (1983). All these results use Kolmogorov complexity and illustrate the power of this method.

Here we will focus on the part of the proof for the case $k = 3$ that uses a Kolmogorov complexity argument. A previous version of the proof by Li using a counting argument was very lengthy and even more complicated as Li states in [2]. Unfortunately, we were not able to find the manuscript that has that proof in order to compare them. However, here we present a part of the later proof that is very crucial and demonstrates the power of Kolmogorov complexity.

We need to define some notation first. A 3-DFA $M = \langle \Sigma, Q, \delta, q_0, F, 3 \rangle$, is a deterministic finite automaton with 3 one-way read-only heads h_1, h_2, h_3. Σ, Q, δ, q_0, and F, are the alphabet, the set of states, the transition function, the starting state, and the set of final states, respectively. M has a one-way read-only input tape. Referring to the language $SUBSTR$ that we defined, x is the *pattern*, y is the *text*, and w.l.o.g. we assume that $\Sigma = \{0,1\}$. At each step, depending on the current state and the ordered k-tuple of symbols seen by the heads h_1, h_2, h_3, M deterministically changes state and moves some of the heads one position to the right. On each input all heads will eventually reach and stay at the final \$ sign. M accepts an input if it is in a final state when all heads reach the final \$ sign. We assume that the heads can detect their coincidences (they can see each other) and that they do not change their relative positions. A lemma that is crucial in the proof for the 3-DFA, M, along with its proof that uses the incompressibility method is the *Moving Lemma* which is presented below.

Graus and Li prove by contradiction that three one-way head DFA cannot detect the existence of a pattern x in the text y. They assume that a 3-DFA M accepts $SUBSTR$. They fix a long enough random string X of length $n^2 + n$, where $n \gg 2^{|Q|}$. They need n to be large enough such that all formulae that they use make sense. They choose X to be of the form $X = xaa_1a_2 \ldots a_{n-1}$ such that for all $i : |x| = |a| = |a_i| = n$. They use x as the pattern of the input and they use x and the as to construct the text part that they will use for the contradiction. The contradiction is that by making the assumption that M accepts $SUBSTR$ they get that $K(X) < |X|$.

The moving lemma that we present here excludes the case that while head h_i crosses a block a of length n in the text such that $K(a) \geq n - O(\log n)$ no other head moves and some of them start moving after h_i crosses a but before h_i reaches the end of input or meets with some other head. Initially all heads are at the \$ sign. One head must first move passing #. Graus and Li use the moving lemma to show that with input $x\#a^{2n}x \ldots h_2$ must pass the # sign when h_1 reaches the end of a^{2n}. If this is not true h_1 would move to the end of input while h_2, h_3

stay stationary by the moving lemma. Then on input $x\#a^{2^n}x$, h_1 first reaches the end before h_2 reaches the # sign. Then when h_2 passes through a^{2^n}, again by the moving lemma, either h_2 is out of pattern x or h_2 moves to the end while h_3 stays stationary. In either case x is not checked.

Lemma 1 *Let a be a block of length n in the text, such that $K(a) \geq n - O(\log n)$. Assume that the 3-DFA M is in state q and one of the heads $h_i, 1 \leq i \leq 3$, is at the first bit of a. Then either*

1. *while h_i crosses a, some other head moves at least one step, or*
2. *h_i will move until it reaches end of input or meets with some other head, while all other heads stay stationary.*

Proof.[using Kolmogorov complexity][M. Geréb-Graus and M. Li] If (2) is false then some other head moves before h_i reaches end of input or meets with some other head. What we want to prove is that some other head moves while h_i crosses a. Assume the contrary (i.e. that no other head moves while h_i crosses a). **The key of the proof** is that if h_i passes a without any other head moving, a can be significantly compressed using M.

From each state of M there is a sequence of length at most $|Q|$ that will make some other head move; otherwise there is a loop. This holds because M has only $|Q|$ states and the shortest distance from any state to any other state is at most $|Q|$ if there is a path at all.

So far we know that a satisfies the following properties:

- No other head moves while h_i crosses a.
- At each bit of a there is a certain sequence of tape symbols of length $|Q|$ starting from this bit that cannot occur since this sequence will lead to the movement of another head (so we have at most $2^{|Q|} - 1$ sequences that can occur).
- At any step, M has to be in a state such that there is a string of length $|Q|$ which would lead to the movement of other heads (otherwise there is a loop).

The number of strings of length $|a|$ that satisfy the above properties is at most:

$$\underbrace{(2^{|Q|}-1)(2^{|Q|}-1)\ldots(2^{|Q|}-1)}_{\frac{|a|}{|Q|}} = (2^{|Q|}-1)^{\frac{|a|}{|Q|}}$$

This is easy to see if you notice that $|a| > |Q|$ and that we cannot start every possible sequence of tape symbols of length $|Q|$ at each one of the $|a|$ bits of a but instead only at multiples of $|Q|$.

We can now take advantage of the small number of strings of length $|a|$ that satisfy the above properties and code a as the ith string in this subset of strings of length $|a|$. In order to describe the index i we need only

$$\log(2^{|Q|}-1)^{\frac{|a|}{|Q|}} = (1-\epsilon)|a|$$

bits, where ϵ is a fixed constant depending only on $|Q|$. So, the information that is needed in order to describe a is:

- $(1-\epsilon)|a|$ bits to describe the index i,
- $O(1)$ bits to describe M,
- $O(1)$ bits to describe the current state of M, and
- $O(1)$ bits for the current bits read by other heads.

The total is less than $K(a)$ which is a contradiction. ■

4 Average Case Analysis of Algorithms

The average case complexity analysis of some algorithms is very difficult. In this section we compare three approaches of the average case analysis of Heapsort, the probabilistic, the counting argument, and the Kolmogorov complexity argument.

Heapsort was discovered in 1964 (by J. W. J. Williams and later improved by R. W. Floyd). It is a fundamental algorithm for sorting, and the algorithm is simply stated and implemented. However, the analysis of its average case complexity was first obtained by R. Schaffer and R. Sedgewick in 1991 [13] [14] (using a counting argument). All previous attempts to analyze its average case performance, using standard probabilistic techniques, were without result because the algorithm does not "preserve randomness" (this will become clear in a while).

Given an array $A[1..n]$ to sort, Heapsort first transforms the keys of the array into a heap (*Build-heap* operation). The heap is then sorted by repeatedly swapping the root of the heap with the last key in the bottom row, and then sifting this new root down to an appropriate position to restore heap order (*Heapify* operation). In order to study the precise performance characteristics of Heapsort and in particular the average case complexity, the primary quantity we have to analyze is the **number of key moves**.

Consider the heap construction process as the reverse of the sorting process. Let $f(n)$ be the number of heaps of n distinct elements. Since the elements are distinct, we can assume without loss of generality that they are from a permutation. The formula that is given by Knuth [7] for $f(n)$ (i.e. the number of ways to label any tree with integers 1 through n such that every node is larger than its two sons) is $n!$ divided by the product of all the subtree sizes. Table 1

gives the exact value of $f(n)$ for small n. The independence of the subheaps can be used to prove that the bottom-up heap construction process preserves randomness. If each of the $n!$ permutations is equally likely before the construction process, then each of the $f(n)$ heaps is equally likely afterwards. Thus, the heap construction process preserves randomness which makes it possible to derive accurate formulas for the average case performance of this process. However, the Heapify procedure does not preserve randomness; simple counting says that the sorting procedure cannot preserve randomness because successive elements in Table 1 do not divide. If each of the 3 heaps of 4 elements are equally likely before the first sorting step, how could each of the 2 heaps of 3 elements be equally likely afterwards? So, after the first step that changes the heap we no longer have the property that all heaps are equally likely as we have assumed at the beginning. Any attempt to find an exact expression for the average would have to overcome this fundamental difficulty.

n	1	2	3	4	5	6	7	8
$f(n)$	1	1	2	3	8	20	80	210

Table 1: $f(n)$ for small n.

Another drawback of the probabilistic approach is that one has to consider expectations and variances and this may be difficult for complicated algorithms or algorithms that use complicated data structures (like the heap structure in Heapsort). Instead, using Kolmogorov complexity one has to reason about a single incompressible object and this fact makes things easier. Also, all statistical properties hold with certainty for this particular object (because it is incompressible) while these properties hold only with high probability in the probabilistic analysis (which makes proofs more complicated and lengthy).

Now we will present the proof using Kolmogorov complexity of the following theorem to illustrate the elegance and the simplicity of this approach. We will focus on the average number of key movements which is the same for both versions of Heapsort (Williams' method and Floyd's method). These two versions differ on the number of comparisons that they perform. In order to study their average case behavior the calculation of the average number of key moves is crucial. This will become obvious later.

This solution is suggested by Ian Munro (it is presented in [9]). The **main idea of the proof** is the following: From a Kolmogorov random permutation of the keys (as almost all permutations are) a lower bound on the Kolmogorov complexity of the corresponding heap is derived. Then, an upper bound on the Kolmogorov complexity of this heap is given based on the description of all heap rearrangements during the sorting process. The average path length is a component of this description, and as a result, a lower bound on the average path length and consequently a lower bound on the average number of key moves is derived. This lower bound is $n \log n - O(n)$. However, in the worst case, Heapify needs to travel to the bottom of the heap on each call so that an upper bound on the number of key moves is

$$\sum_{1 \le i \le n} \lfloor \log(n/i) \rfloor + \sum_{1 \le i \le n} \lfloor \log(i) \rfloor = n \log n + O(n) \tag{1}$$

The first term is due to the key moves during the Build-heap process and the second term is due to the key moves that take place during the last n Heapify operations that sort the heap by repeatedly sifting the new root (which was the last key in the bottom row before the swapping) down to an appropriate position to restore the heap property.

Theorem 3 *On the average over all initial permutations of the keys, Heapsort makes at least* $n \log n - O(n)$ *key moves.*

Proof.[using Kolmogorov complexity][Ian Munro] Given n keys, choose a permutation p of these keys (out of the $n!$ ($\approx n^n e^{-n}\sqrt{2\pi n}$) permutations that exist), such that

$$K(p|n) \geq n \log n - 2n \tag{2}$$

A permutation like this exists because of Theorem 2 (with $m = n!$).

Consider the heap h that is built by the Heapify step with input p that satisfies Equation 2. Then

$$K(h|n) \geq n \log n - 6n \tag{3}$$

This can be proved by contradiction. Assume that $K(h|n) < n \log n - 6n$. Then p can be described using h and n in less than $n \log n - 2n$ bits. We do this be encoding the Heapify process which constructs h from p. In each iteration when we push the key k down the subtree, we record the path which key k traveled: 0 for a left branch, 1 for a right branch and 2 for halt. We need $n \log 3 \sum_i i/2^{i+1}$ bits in total because there are n keys, 3 choices to record at each step (i.e. $\log 3$ bits for each) and the average length of a path is $\sum_i i/2^{i+1}$ (there are 2^{i+1} paths of length i). Note that $n \log 3 \sum_i i/2^{i+1} \leq 2n \log 3$. Now, using the final heap h and the above description of the Heapify process we can reconstruct p by reversing the procedure of Heapify. Thus, $K(p|n) < K(h|n) + 2n \log 3 + O(1) < n \log n - 2n$, reaching a contradiction. This completes the proof of a lower bound on the Kolmogorov complexity of a heap that corresponds to a Kolmogorov random permutation of the keys.

Now let us prove an upper bound on the Kolmogorov complexity of the heap. We can have the history of the $n-1$ heap rearrangements during the sorting process by recording for $i := n-1, \ldots, 2$, the final position where $A[i]$ is inserted into the heap. We encode the final position by describing the path from the root to the position (using 0 and 1 to denote left and right branches respectively). Each path is encoded as (a, l, s) where $a = 0$ means that $l = l(s)$ and $a = 1$ means that $l = \log n - l(s)$, $l(s)$ is a self-delimiting binary form of s, and s is a sequence of 0's and 1's encoding the actual path. If the ith path has length d_i, then this description for the path requires at most

$$1 + d_i + \min\{2 \log(\log n - d_i), 2 \log d_i\} \tag{4}$$

bits because we need 1 bit to represent a, d_i bits to represent the sequence s and $\min\{2 \log(\log n - d_i), 2 \log d_i$ to represent the self-delimiting binary form of either $l(s)$ or $\log n - l(s)$, whichever is shorter. Finally, we concatenate the descriptions of all these paths into one sequence H.

The heap h can be described using the following information:
1. this discussion,
2. the sequence H, and
3. the number n.

We can reconstruct the heap h from this description by starting with a sorted list and simulate the sorting process in reverse. So, $K(h|n) \leq l(H) + O(1)$ and by Equation 3 we have $l(H) \geq n \log n - 6n$. By Equation 4 this is only possible if the average path length is at least $\log n - c'$, for some fixed constant c'. This is the average number of key moves in each iteration of the sorting process. Multiplying by n we have that Heapsort performs at least $n \log n - O(n)$ key moves starting with heap h, which corresponds to a random permutation p. Since almost all permutations are Kolmogorov random, this bound holds for almost all permutations *on the average*. ■

Now that we can calculate the average depth, d, of the final position of a key k which actually is the average number of key moves that Heapsort algorithm performs, we can compare the average behavior of the two versions of Heapsort. But first we have to calculate the number of comparisons for these two versions (Williams' and Floyd's method). Williams' method uses $2d$ comparisons, and Floyd's method uses $d + 2\delta$ comparisons, where $\delta = \log n - d$ [9] (the heap is of depth $\log n$). The difference between the two methods lays on how *Heapify* operates. Williams' method goes from the root down the heap. It makes two comparisons with the son nodes and one key movement at each step, until the key k reaches its final position. Floyd's method first goes from the root down the heap to a leaf, making only one comparison in each step. Then, it goes from the bottom of the heap up the tree, making one comparison at each step until it finds the final position for key k. Then it moves the keys shifting every ancestor of k one step up the tree. Although the final positions in the two methods are the same the number of comparisons differ as we discussed before. By substituting $n \log n - O(n)$ for d in the formulae for the number of comparisons for the two methods we have that Heapsort with Williams' method makes $2n \log n - O(n)$ key comparisons on the average, and Heapsort with Floyd's method makes $n \log n + O(n)$ key comparisons on the average which completes the average case analysis of Heapsort. Note how crucial was the analysis of the average number of key moves in the average case analysis of Heapsort.

We now present a sketch of the proof using a counting argument by R. Schaffer [14] of the following theorem on the average number of key moves required by Heapsort. The proof involves a lot of definitions. The counting arguments that are used are very complicated and lengthy. Because of the complexity of the arguments, the proof lacks intuitiveness. Before we sketch out the main steps of the proof we give any necessary definitions.

Theorem 4 *The average number of key moves required to sort using Heapsort a random permutation of n distinct keys is $\sim n \log n$.*

Sketch of the proof. [using a counting argument][R. Schaffer] Let H_n be a heap of

size n (i.e. with n keys). To avoid confusion it should be noted that heap construction for this context means starting with a heap on a single key and adding progressively larger keys until a heap of size n is reached. Notice that this is different from building a heap with the Build-heap operation. Recall that the Heapify (or *Siftdown*) operation produces a heap H_{n-1} from a heap H_n. Given a heap H_{n-1}, it is convenient to consider working backwards to generate all heaps H_n that yield that heap after a Siftdown operation. We call the operation that reverses a step of the Heapsort process by transforming a heap H_{n-1} to a heap H_n, a *Pulldown* operation. Thus, Pulldown and Siftdown are inverses of each other. Pulldown would have to take the key from some position $A[k]$ and place it in position $A[n]$ at the bottom, fill the vacancy at $A[k]$ by shifting down all keys above it in the heap and assign the nth input key to the root (note that this key is the largest so far because we add progressively larger keys). The position k is the parameter of Pulldown. Every heap can be associated with a unique sequence of pulldowns (which is described as the corresponding sequence of the parameters of the pulldowns). Running Heapsort results in a sequence of progressively smaller heaps. The reverse of such a progression corresponds to a unique Pulldown sequence. Now the reason for adding progressively larger keys when we perform heap construction by repeated pulldowns becomes clear; this procedure is the reverse operation of using repeated siftdowns to sort the keys where the largest element is taken out from the root each time.

Pulldown sequences are crucial to the analysis of the average case of Heapsort because the cost of sorting a heap is closely related to the cost of constructing the heap by repeated pulldowns. In order to take advantage of this correspondence, Schaffer and Sedgewick measure the cost of each pulldown in a way that is directly related to the cost that will be incurred by Heapsort when reversing that pulldown. Given a pulldown sequence, $P = \{p_i\}_{i=2}^{n}$, they consider four quantities, namely $B_W(p_i)$, $C_W(p_i)$, $B_F(p_i)$, $C_F(p_i)$, where W stands for the Williams' version of Heapsort and F stands for the Floyd's version. In order to analyze the average number of key moves required by Heapsort, they have to consider the two versions of Heapsort from the beginning although the key stops at the same position in the heap whether it is being moved down by Williams' or Floyd's Heapsort which means that both versions generate the same sequence of progressively smaller intermediate heaps when sorting a list. Recall that this is not necessary in the proof using the Kolmogorov complexity argument!

The B_W cost of pulldown p_i, $B_W(p_i)$, is defined as the level containing the position p_i (where the root is considered to be at level 1), the C_W cost of pulldown p_i, $C_W(p_i)$, is defined as the number of times the right child is selected along the path from the root to position p_i, and $B_F(p_i)$ and $C_F(p_i)$, are the corresponding quantities for the Floyd's version of Heapsort. The cost of a pulldown sequence (and the cost of the corresponding heap) is defined as the sum of the costs of the pulldowns.

In order to determine the asymptotic complexity of sorting a random heap structure, it is sufficient to determine the expected B_W, C_W, B_F, and C_F costs of the heap constructed by a random pulldown sequence. The proof shows that given a uniformly generated random permutation on n keys in $A[1 \ldots n]$, if H is the heap that is built from that permutation, then it is expected that $B_W(H) \sim n \log n$, $C_W(H) \sim \frac{1}{2} n \log n$, $B_f(H) = O(n)$, and $C_F(H) \sim \frac{1}{2} n \log n$.

Here, we sketch out only the proof for the $B_W(H)$. The average B_W cost of a random heap structure is determined by showing that all but an asymptotically negligible number of pulldown sequences build heaps that have costs near the average.

First, Schaffer proves the fact that for any n, the number of heaps $|H_n|$ is $|H_n| > n!/4^n$ (he uses this fact later to prove that only an exponentially small fraction of the heaps in H_n have abnormal costs). Then, given positive integers n and t, for sufficiently large n, he finds an upper bound on the number of B_W cost sequences C having cost equal to t. He proves this upper bound using a very complicated and lengthy counting argument. In the rest of the proof he shows that the average pulldown sequence P has B_W cost at least $n \log n + O(n)$. Given a B_W cost sequence, C, he finds an upper bound on the number of the pulldown sequences for which $B_W(P) = C$. Then he finds a bound on the number of pulldown sequences with $B_W(H(P)) \leq T$ that is exponentially smaller than $|H_n|$ for some reasonably large value of T. He chooses as T the quantity $n(\lceil \log(n+1) \rceil - 10)$ and he finds the upper bound $U = O((\frac{n}{15})^n)$. Then it is easy to see that the average of $B_W(H(P))$ over all pulldown sequences P is greater than:

$$\begin{aligned} \frac{(|H_n| - U)T + U0}{|H_n|} &= \frac{(|H_n| - O((n/15)^n))n(\lceil \log(n+1) \rceil - 10) + O((n/15)^n)0}{|H_n|} \\ &> n(\lceil \log(n+1) \rceil - 10)(1 + O((4e/15)^n)) \end{aligned}$$

which completes the proof for the $B_W(H(P))$. Schaffer uses similar arguments to determine the expected values of $C_W(H(P))$, $B_F(H(P))$, and $C_F(H(P))$. ■

5 Formal Languages

Kolmogorov complexity can also be used to prove that certain languages are not regular or not context free. We believe that in comparison to the pumping lemmas, which is the usual way to prove such things, Kolmogorov complexity arguments are more intuitive, elegant and simpler. In the following subsections we present the Kolmogorov complexity replacements of the pumping lemmas for regular languages and for deterministic context free languages and we give several examples to demonstrate the advantages of this approach over the traditional approach with the pumping lemmas. For completeness we state here that this approach is also successful at the high end of the Chomsky hierarchy since one can quantify nonrecursiveness in terms of Kolmogorov complexity (see [8] for more details).

5.1 Regular Languages

In comparison to the pumping lemma for regular languages, Kolmogorov complexity arguments are more intuitive, more elegant, and simpler. One more reason that makes Kolmogorov complexity arguments preferable to the pumping lemmas is the following: Pumping lemmas are

needed because it is sometimes hard to use the characterizations of regular languages (like the Myhill-Nerode Theorem) in order to show non-regularity. These characterizations are mainly practically useful to show regularity. At the end of this subsection we will present the compressibility characterization which serves both purposes (i.e. prove regularity and non-regularity). In this sense the Kolmogorov complexity interpretation of the Myhill-Nerode Theorem, can be applied whenever the pumping lemma applies and also in situations where pumping lemmas do not apply. This interpretation is stated as follows:

Lemma 2 [Kolmogorov-Complexity-Regularity (KCR)] [8] *Let L be a regular language. Then for some constant c depending only on L and for each string x, if y is the nth string in lexicographical order in $L_x = \{y : xy \in L\}$ (or in the complement of L_x) then $K(y) \leq K(n) + c$.*

Proof. Let L be a regular language. A string y such that $xy \in L$, for some x and n as in the Lemma, can be described by:

1. this discussion, and a description of the FA that accepts L,
2. the state of the FA after processing x,
3. the number n.

■

In order to demonstrate the advantages of using the KRC lemma instead of using the pumping lemma we present here the two kinds of proof for some demonstrative examples. But first, let us state formally the pumping lemma for regular languages:

Lemma 3 [Pumping Lemma][Y. Bar-Hillel, M. Perles, and E. Shamir [1961]] *Let L be a regular set. Then there is a constant n such that if z is any word in L, and $|z| \geq n$, we may write $z = uvw$ in such a way that $|uv| \leq n$, $|v| \geq 1$, and for all $i \geq 0$, uv^iw is in L.*

Fact 6 [Exercise 3.1c in [3]] *The language $L = \{0^n : n \, is \, prime\}$ is not regular.*

Proof.[using the pumping lemma] Assume L is regular and let p be the smallest prime which is greater than or equal to n where n is the integer in the pumping lemma. Let $z = 0^p$. The string z is in L. By the pumping lemma, 0^p can be written as uvw, where $1 \leq |v| \leq n$ and uv^iw is in L for all i. Also $|uv| \leq n$. Let $i = p + 1$. By the pumping lemma, the string $uv^{p+1}w$ must be in L. However, this string can be written as $uvwv^p = 0^pv^p$ because the parts u, v, w are all just strings of 0's. Say that v consists of m 0's (i.e. $|v| = m$). Then the string 0^pv^p can be written as $0^{p+pm} = 0^{p(m+1)}$ which obviously is not in L, thus reaching a contradiction. ■

Proof.[using the KCR lemma][M. Li and P. M. B. Vitányi] Assume L is regular. Consider the string xy, of the KCR lemma, consisting of p 0's where p is the $(k+1)$th prime. Set in the lemma x equal $0^{p'}$ with p' the kth prime, so $y = 0^{p-p'}$, and $n = 1$. It follows that $K(p - p') = O(1)$. However , the differences between the consecutive primes rise unbounded so there is an unbounded number of integers of Kolmogorov complexity $O(1)$. Since there are only $O(1)$ descriptions of length $O(1)$, we have a contradiction. ■

Fact 7 [Exercise 3.6* in [3]] *The language* $L = \{0^i 1^j : \gcd(i,j) = 1\}$ *is not regular.*

Proof.[using the pumping lemma][obtained with K. Kalpakis] Assume L is regular and let p be the smallest prime which is greater than or equal to n where n is the integer in the pumping lemma. Let $z = 0^p 1^{(p-1)!}$. The string z is in L because $\gcd(p,(p-1)!) = 1$. By the pumping lemma, $0^p 1^{(p-1)!}$ can be written as uvw, where $1 \leq |v| \leq n$ and $uv^i w$ is in L for all i. Also $|uv| \leq n$ and in particular uv is a prefix of 0^p. Let $i = 0$. Then the string $0^a 1^{(p-1)!}$, where $a < p$, is not in L because $\gcd(a,(p-1)!) \neq 1$ for all $2 \leq a < p$. This is easy to see if you consider $(p-1)!$ as $1 \cdot 2 \cdot 3 \cdot 4 \cdots (p-1)$. ■

Proof.[using the KCR lemma][M. Li and P. M. B. Vitányi] Set $x = 0^{(p-1)!}1$, where $p > 3$ is a prime, length of p, $l(p) = n$ and $K(p) \geq \log n - \log\log n$. Then, the lexicographically first word in L_x is 1^{p-1}, contradicting the KCR lemma. ■

Proof.[using the KCR lemma] Choose the string x of the KCR lemma to be 0^p where p is any prime. Then, 1^p is the lexicographically first nonempty word in $\overline{L_x} \cap 1^*$. Thus, by the KCR lemma, there is a constant c such that for all p we have $K(p) < c$, reaching a contradiction. ■

We now present for completeness the compressibility characterization of formal languages (the reader is referred to [9], [8] for more information). While the pumping lemmas are not precise enough to characterize the regular languages, with Kolmogorov complexity this is easy. The KCR Lemma is a direct corollary of the characterization below.

First we need some notation. Enumerate $\Sigma^* = \{y_1, y_2, \ldots\}$ with y_i the ith element in the total order. For $L \subseteq \Sigma^*$ and $x \in \Sigma^*$ let $\chi = \chi_1 \chi_2 \ldots$ be the *characteristic sequence* of $L_x = \{y : xy \in L\}$, defined by $\chi_i = 1$ if $xy_i \in L$ and $\chi_i = 0$ otherwise. We denote $\chi_1 \ldots \chi_n$ by $\chi_{1:n}$.

Theorem 5 *(***Regular Kolmogorov Complexity Characterization***) Let* $L \subseteq \Sigma^*$. *There is a constant* c_L *depending only on* L, *such that the following statements are equivalent:*
(i) L *is regular*
(ii) for all $x \in \Sigma^*$, *for all* n, $K(\chi_{1:n}|n) \leq c_L$
(iii) for all $x \in \Sigma^*$, *for all* n, $K(\chi_{1:n}) \leq K(n) + c_L$
(iv) for all $x \in \Sigma^*$, *for all* n, $K(\chi_{1:n}) \leq \log n + c_L$

5.2 Deterministic Context-Free Languages

M. Li and P.M.B. Vitànyi, present a Kolmogorov complexity approach for deterministic context free languages (DCFLs) ([8]). The well-known pumping lemma for context free languages (CFLs) can be used to prove that a certain language is a non-DCFL only if this language is also non-CFL (e.g. the language $\{xx : x \in \Sigma^*\}$). However, there are languages that are CFLs but not DCFLs (e.g. $\{xx^R : x \in \Sigma^*\}$). There exist special pumping lemmas for DCFLs [18] but usually they are very difficult to use. The following lemma using Kolmogorov complexity is simpler, more intuitive and more powerful. To illustrate these properties we present an example of a context free language that has proved to be not DCFL only with great effort before but which has a very elegant and simple proof of not being a DCFL using this lemma. Actually, below we present (without proof) a corollary which is an easier but more restricted version of the lemma.

Corollary 1 [Kolmogorov Complexity-DCFL (KC-DCFL)] [8] *Let $L \subseteq \Sigma^*$ be a DCFL and let c be a constant. Let x and y be fixed finite words over Σ and let ω be a recursive sequence over Σ. Let u be a suffix of $yy \ldots yx$, let v be a prefix of ω, and let $w \in \Sigma^*$ such that:*
(i) v can be described in c bits given L_u in lexicographical order;
(ii) w can be described in c bits given L_{uv} in lexicographical order; and
(iii) $K(v) \geq 2 \log\log l(u)$
where L_u is defined as $L_u = \{z : uz \in L\}$ and L_{uv} is defined as $L_{uv} = \{z : uvz \in L\}$. Then, there is a constant c' depending only on L, c, x, y, ω such that $K(w) \leq c'$.

where a one-way infinite string $\omega = \omega_1\omega_2 \ldots$ over Σ is *recursive* if there is a total recursive function $f : N \to \Sigma$ such that $\omega_i := f(i)$ for all i. Intuitively, the Corollary tries to capture the following: if v is the first word in L_u, then processing the v part of input uv must have already used up the information of u. But if there is not much information left on the pushdown store, then the first word of w in L_{uv} cannot have high Kolmogorov complexity.

Fact 8 [Exercise **10.5a in [3]] *The language $L = \{xx^R : x \in \Sigma^*\}$ is not DCFL.*

Proof.[using the pumping lemma for CFLs and closure properties of DCFLs][solution in [3] p.266] Suppose L were a DCFL. Then by closure of DCFLs under intersection with regular sets we have $L_2 = L \cap (01)^*(10)^*(01)^*(10)^*$ be a DCFL. Language L_2 can be expressed as:

$$\{(01)^i(10)^j(01)^j(10)^i | i, j \geq 0, i \text{ and } j \text{ not both } 0\}$$

Using the closure of DCFLs under the operation MIN defined as $MIN(L) = \{\ x|x$ is in L and no w in L is a proper prefix of $x\ \}$, we have that $L_3 = MIN(L_2)$ is a DCFL. But $L_3 = \{(01)^i(10)^j(01)^j(10)^i | 0 \leq j < i\}$, since if $j \geq i$, a prefix is in L_2. Let h be the homomorphism $h(a) = 01$ and $h(b) = 10$. Then $L_4 = h^{-1}(L_3) = \{a^ib^ja^jb^i | 0 \leq j < i\}$ is a DCFL by closure of DCFLs under inverse homomorphism. The pumping lemma for CFL with

$z = a^{n+1}b^n a^n b^{n+1}$ shows that L_4 is not even a CFL. ■

Proof.[using the KC-DCFL Corollary][based on a similar proof of the language $\mathbf{L} = \{\mathbf{x} : \mathbf{x} = \mathbf{x^R}, \mathbf{x} \in \Sigma^*\}$ being not a DCFL by M. Li and P.M.B. Vitányi] Assume the contrary. Set $u = 0^{2n}11$ and $v = 0^{2n}$, $K(n) \geq \log n$, satisfying (iii) of the KC-DCFL Corollary. Since v is lexicographically the first word in L_u, item (i) of the KC-DCFL Corollary is satisfied. The lexicographically first nonempty word in L_{uv} is 110^{2n}, and so we can set $w = 110^{2n}$ satisfying item (ii) of the Corollary. But now we have $K(w) = \Omega(\log n)$, contradicting the KC-DCFL Corollary. ■

6 Kolmogorov complexity and random graphs

Kolmogorov complexity has been defined for strings so in order to use it in proving properties of graphs either we have to encode graphs to binary strings and then apply the incompressibility method as before or we have to extend the Kolmogorov complexity theory such that it applies to more generalized objects such as graphs. The Kolmogorov complexity proof that we present here follows the first approach which is the obvious one for the type of problem that we consider.

We present a proof of the fact that almost every labeled tree on n vertices has maximum degree $O(\log n / \log\log n)$. The proof involves exhibiting an algorithm for producing a string and then showing that such an algorithm could not produce an incompressible (or Kolmogorov-random) string. It also uses the method of deleting blocks of bits from Kolmogorov-random strings that we presented in Section 3. By coding trees with binary strings (that is, by establishing a one-to-one correspondence between trees and strings), it is shown that if a tree has "large" maximum degree, then its code is not very "random" in the Kolmogorov sense. Since "almost all strings" are random, it follows that almost every tree does not have large maximum degree.

The original proof given by J.W. Moon [11] uses probability theory and combinatorial (counting) arguments ([11]). This proof is more complicated and more lengthy than the Kolmogorov complexity proof. We present here the basic steps just to give the reader a flavor of the complexity of the proof.

Sketch of the proof.[using probabilistic and counting arguments][J.W. Moon] First the fact that there are

$$\binom{n-2}{k-1}(n-1)^{n-k-1}$$

trees T_n with n labeled nodes in which the degree $d(x) = k$ for each node x is proven. Then, a tree T_n is picked at random from the set of the n^{n-2} trees with n labeled nodes and the degree $d(x)$ of an arbitrary node x is considered. The probability $\Pr\{d(x) > k\}$ is proven to be $\Pr\{d(x) > k\} < \frac{1}{k!}$. Then from Boole's inequality, the result, $\Pr\{D > k\} \leq \frac{n}{k!}$ follows, where D

is the maximum degree of the nodes in T_n. As a corollary of the previous result we have that

$$D \leq (1+\epsilon)\frac{\log n}{\log\log n}$$

for almost all trees T_n, that is, for all but a fraction that tends to zero as n tends to infinity, (ϵ is a positive constant). ■

The proof that we present below uses a Kolmogorov complexity argument [6]. It establishes the result for trees on n vertices where n is a power of 2 (this assumption can be removed easily by a combinatorial argument). Fundamental to this proof is the notion of a Prüfer code for a tree. Prüfer in his proof of Cayley's theorem regarding the number of labeled trees on n vertices, showed that there exists a 1-1 correspondence between $(n-2)$-length sequences of integers from the set $\{1,2,\ldots,n\}$ and labeled trees on n vertices. Further, if an integer k occurs exactly m times in a sequence corresponding to a tree T, then the vertex in T with label k has degree $m+1$.

Fact 9 *For almost every tree, T, on n vertices where n is a power of 2, the maximum degree* $\Delta(T) = O(\log n/\log\log n)$.

Proof.[by Kolmogorov complexity][W. W. Kirchherr] In general, in order to prove that a property holds for almost every tree, it suffices to prove that every tree encoded by a g-incompressible string has that property because "almost all" strings are g-incompressible. Here, we assume $g(n) = d\log n$ for some $d \geq 1$. Take a g-incompressible string S of length $N = (n-2)\log n$. S can be seen as an encoding of $n-2$ integers (so it consists of $n-2$ blocks) each one in the range $0\ldots(n-1)$. Let S code tree T_S. Assume $\Delta(T_S) = k$. This means that some block B (of length $\log n$) appears $k-1$ times in the string S. The **key of the proof** is to show that if k is too large, then S is not g-incompressible.

Let $\overline{S}$ be the string S with the $k-1$ appearances of the block B removed. Then S can be produced from $\overline{S}$ if (1) a single copy of the block B, and (2) the locations of the missing blocks, are provided. This information needs to be specified in a self-delimiting form so that the beginning and end of each input can be distinguished. Recall that a self-delimiting encoding of x uses about $|x| + 2\log|x|$ bits so an integer a can be encoded in about $\log a + 2\log\log a$ bits.

Consider the blocks in S numbered $1,2,\ldots,n-2$ and let $p_1, p_2, \ldots, p_{k-1}$, be the numbers of the $k-1$ copies of block B in S. By specifying p_1 and $p_i - p_{i-1}$ for $2 \leq i \leq k-1$ the numbers of the $k-1$ missing blocks in $\overline{S}$ can be specified with $C = (k-1)[\log(n/k-1) + 2\log\log(n/k-1)]$ bits. Here, we use the fact that $\sum_{i=1}^{r}\log m_i \leq r\log(n/r)$ where m_i, $1 \leq i \leq r$, are positive integers. Thus, for the Kolmogorov complexity of S, $K(S)$, we have:

$$\begin{aligned} K(S) \quad \leq \quad & [N - (k-1)\log n + 2\log(N-(k-1)\log n)] \\ & +[\log n + 2\log\log n] + C \end{aligned}$$

where the first term is the number of bits to encode $\overline{S}$ and the second term is the number of bits to encode a single copy of the block B. Since $K(S) \geq N - g(n)$ and using $N = (n-2)\log n$ we have (ignoring constant terms):

$$(k-1)\log(k-1) + 2(k-1)\log\log(k-1) \leq 3\log n + g(n) + (2k+2)\log\log n + k$$

which is true only for $k = O(\log n / \log\log n)$. ■

6.1 Comments and further research

We believe that it is easier to use Kolmogorov complexity arguments rather than probability theory and counting arguments when we want to prove that a certain property holds for almost all objects of a set. Also the steps of applying the incompressibility method of Kolmogorov complexity are well defined and the proof is easy to follow. However, in order to apply this method to objects which are not strings we need to encode the objects to strings and the encoding should "contain", in some sense, information about the property we want to prove. Notice, for example, the use of Prüfer encoding for trees in the previous proof of an upper bound on the maximum degree of the nodes of a labeled tree. The degree k of a node x is represented by the number of occurrences of the label of node x in the Prüfer encoding of that tree. Also, the largest the maximum degree for a tree, the more occurrences of a node's label in the code and the less random the code is in the Kolmogorov sense.

We can apply the same idea in proofs of properties that hold for almost all trees or graphs in general. One such proof of a property of binary trees that we are working on is the following: The average height of binary trees with n internal nodes is asymptotic to $2\sqrt{\pi n}$. This result has been proved by P. Flajolet and A. Odlyzko ([1]) using a very complicated and lengthy argument based on the singularity analysis of generating functions. A similar result for general rooted trees is that the expected value of their height is asymptotically equal to $\sqrt{2\pi n}$. This result is due to A. Rényi and G. Szekeres ([12]). An appropriate encoding for binary trees with n nodes for this problem may be the well-formed bracket sequences with n pairs of brackets. Let us associate with the left and the right parenthesis the symbols "0" and "1" respectively. Then, after a sequence of appropriate rotations of some subtrees of the tree (which do not change the height of the tree of course) we can have a sequence of consecutive 1's in the code of the tree, representing the height of the tree.

We believe that there are a lot of properties of random trees and graphs that can be proved easily using Kolmogorov complexity. For a review of properties of random graphs the reader is encouraged to look at [5]. Another approach that would be more appealing and could solve a lot of problems in random graphs is the extension of Kolmogorov complexity to more generalized objects than strings as we mentioned before. Also, as far as we know, not so much work has been done in this area. Although the notion of Kolmogorov complexity has been defined for arbitrary objects in [16] we find it not very easy to use. Using a special form of recursive topological spaces, called partition spaces, they define a recursive topology which uses a level of partition for approximation of arbitrary objects.

7 Conclusions

In this paper we considered applications of Kolmogorov complexity in several different areas such as lower bounds, average case analysis of algorithms, formal language theory, and random graphs. Kolmogorov complexity has not only very abstract use but also can be used in formal proofs of mathematical and computational results. One would say that these proofs are very abstract but this is one of the reasons that make these proofs more intuitive. Although Kolmogorov complexity contains deep mathematics, the amount of mathematics one needs to know to apply the theory in several areas is little.

We compared the incompressibility method of Kolmogorov complexity with other more traditional methods such as proofs by counting, proofs by probabilistic arguments and proofs by pumping lemmas for formal languages. We argued that the Kolmogorov complexity arguments are more elegant, intuitive, simple and less lengthy than the other arguments in most cases. The power of Kolmogorov arguments and their simplicity over the counting arguments is justified by the following fact. In Kolmogorov proofs we deal only with a single, random, "typical" string that exists although it cannot be exhibited or proved to be random or "typical". We use that string to prove that if the property that we want to prove does not hold, then the string can be compressed, reaching a contradiction. All the Kolmogorov complexity proofs that we presented have the same structure and they are based on a standard, simple argument that is applied everywhere (the incompressibility argument). This makes the proofs very easy to follow. All of them are proofs by contradiction and they all rely on the fact that "almost all" strings are incompressible (i.e. Kolmogorov random).

In a lower bound proof by counting *all inputs* have to be considered and this makes the proofs complicated. In an average case analysis of algorithms the probabilistic approach cannot be used when the algorithm does not preserve randomness. Although the probabilistic style allows the direct use of probability theorems which are very powerful, one has to consider expectations and variances and this may be difficult for complicated algorithms or algorithms with complicated data structures. Also, the statistical properties hold only with high probability in the probabilistic approach and this makes the proofs more complicated. Instead, these properties hold with certainty for the particular Kolmogorov incompressible object. In the case of formal languages the Kolmogorov complexity lemmas are more powerful than the pumping lemmas as the simplicity of the proofs using these lemmas illustrates.

Kolmogorov complexity can be used very easily to prove properties that hold for "almost all" objects of a set. However, in order to apply these arguments to objects that are not strings we need to encode the objects to strings and the encoding should "contain", in some sense, information about the property that we want to prove (as was the case with the maximum degree of a labeled tree). An object may be a tree or a graph. There are a lot of properties of random trees and graphs that can be proved more easily using Kolmogorov complexity. Also, the notion of Kolmogorov complexity can be extended to more generalized or arbitrary objects such that the formulation satisfies most of the previous results obtained for sequences of symbols.

Acknowledgments

I thank Professor Yaacov Yesha for his valuable comments, suggestions and for many helpful conversations about this work. I thank Professor Richard Chang for his numerous comments that helped considerably to improve the quality and the readability of the paper. I thank Konstantinos Kalpakis for many helpful discussions on this topic. I am also grateful for the constructive comments of Professor Samuel Lomonaco.

References

[1] P. Flajolet and A. Odlyzko. The Average Height of Binary Trees and Other Simple Trees. *Journal of Computer and System Sciences*, 25:171–213, 1982.

[2] M. Geréb-Graus and M. Li. Three one-way heads cannot do string matching. *Journal of Computer and System Sciences*, 48(1):1–8, 1994.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[4] T. Jiang and M. Li. k One-way Heads Cannot Do String-Matching. In *25th ACM STOC '93-5/93/CA, USA*, May 1993.

[5] M. Karonski. A Review of Random Graphs. *Journal of Graph Theory*, 6:349–389, 1982.

[6] W. W. Kirchherr. Kolmogorov complexity and random graphs. *Information Processing Letters*, 41:125–130, 1992.

[7] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley, 1973.

[8] M. Li and P. M. Vitányi. A New Approach to Formal Language Theory by Kolmogorov Complexity. Preprint, SIAM J. Comput., to appear, 1994.

[9] M. Li and P. M. B. Vitányi. *An introduction to Kolmogorov complexity and its Applications*. Springer-Verlag, 1993.

[10] M. Li and Y. Yesha. String-matching cannot be done by a two-head one-way deterministic finite automaton . *Information Processing Letters*, 22:231–235, 1986.

[11] J. W. Moon. On the maximum degree in a random tree. *Michigan Math. Journal*, 15:429–432, 1968.

[12] A. Rényi and G. Szekeres. On the height of trees. *Journal Austral. Math.*, 7:497–507, 1967.

[13] R. Schaffer and R. Sedgewick. Analysis of Heapsort. Technical Report CS-TR-330-91, Princeton University, 1991.

[14] R. W. Schaffer. *Analysis of Heapsort*. PhD thesis, Princeton University, 1992.

[15] A. L. Selman, editor. *Complexity Theory Retrospective*, chapter 7, pages 147–203. Springer-Verlag, 1990.

[16] A. Shenhar. On the Kolmogorov Complexity of Arbitrary Objects. *Journal of Complexity*, 9:499–517, 1993.

[17] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A, chapter 4, pages 187–254. Elsevier/MIT Press, 1990.

[18] S. Yu. A pumping lemma for Deterministic Context-Free Lenguages. *Information Processing Letters*, 31:47–51, 1989.