Kolmogorov complexity of #P functions

Frederic Green
Jacobo Torán

Report LSI–91–2

Kolmogorov Complexity of #P Functions

Frederic Green[1]
Dept. of Math/CS
Clark University
Worcester, MA 01610
U.S.A.

Jacobo Torán[2]
Departamento L.S.I.
U. Politècnica de Catalunya
Pau Gargallo 5
E-08028 Barcelona

Abstract

Are the outputs of #P functions "random"? We may phrase this question more precisely: Are there #P functions whose outputs cannot in general be compressed into a string of small length, and recovered quickly from that string, also given the input as additional information? We prove that the answer to this question is "yes" if and only if P $\neq$ PP.

1. Introduction

The ability to count solutions, characterized by the function class #P [13], has proved to be unexpectedly powerful. Most remarkable is the result by Toda which states that any set in the polynomial time hierarchy can be computed by a polynomial time machine with access to a function in #P [11].

In this paper we study another aspect of the computational power of #P functions. We try to answer the question of whether the functions in the class #P can produce outputs which are, in some sense, random. The question is motivated by the intuition that if #P is indeed a "hard" class of functions, there ought to be functions in the class that produce complicated outputs. Interestingly, using a natural measure of randomness we find that the question is *equivalent* to the question of whether polynomial time is different than probabilistic polynomial time (P $\neq$ PP). The measure of randomness we use is generalized Kolmogorov complexity.

Intuitively a string is Kolmogorov random if it cannot be compressed (cf. [8]). Generalized Kolmogorov complexity measures how *far* a string can be compressed and how *fast* it can be recomputed from its compression. The first applications of generalized Kolmogorov

[1]The research of this author was supported by a grant from the Dirección General de Investigación Científica y Técnica (DGICYT), Spanish Ministry of Education

[2]The research of this author was supported by the ESPRIT BRA Program of the EC under contract no. 3075, project ALCOM.

complexity to structural complexity issues were due to Hartmanis [4]. Following his article we give the following definition.

Definition 1 Let M_U be a fixed universal Turing machine, and let $s, t : \mathbb{N} \longrightarrow \mathbb{N}$ be two functions. $K[s(n), t(n)] = \{x \mid \exists y \ |y| \leq s(|x|)$ and $M_U(y)$ prints x within $t(|x|)$ steps $\}$

Intuitively, $K[s(n), t(n)]$ defines the set of strings x which can be compressed to size $s(|x|)$ and can be recomputed in time $t(|x|)$.

We can obtain a definition of Kolmogorov complexity of string x relative to another string z by giving z as additional information to the universal machine.

Definition 2 $K[s(n), t(n) \mid z] = \{x \mid \exists y \ |y| \leq s(|x|)$ and $M_U(\langle y, z \rangle)$ prints x within $t(|x|)$ steps $\}$

Observe that for every string z, $K[s(n), t(n)] \subseteq K[s(n), t(n) \mid z]$.

We are especially interested in the set of strings which can be compressed to logarithmic size, and can be recovered in polynomial time, the family of sets $K[c\log(n), n^c]$. We will say that a set A is *Kolmogorov simple* if for some constant c every string in A belongs to $K[c\log(n), n^c]$. We will say that a function f has *Kolmogorov simple images relative to the input* if for some constant c, for each x, $f(x) \in K[c\log(n), n^c \mid x]$.

Generalized Kolmogorov complexity has been used in the past to obtain new characterizations of computational problems. With a different terminology Adleman [1] showed that P=NP iff satisfiable boolean formulas have satisfying assignments which are Kolmogorov simple relative to the formula. If we consider the class of optimization functions *opt*P [7], Adleman's result can be stated in the following way: P=NP iff *opt*P functions have Kolmogorov simple images relative to the input. Another connection between Kolmogorov simple sets and complexity classes was recently proved in [5]. The authors showed that $\mathrm{P}^{\mathrm{NP}}=\mathrm{P}^{\mathrm{NP[log]}}$ iff every language in Δ_2 is accepted by a machine with Kolmogorov simple pronouncements relative to the input.

This article continues this line of research. We show the following

Main Result: #P functions have Kolmogorov simple images relative to the input if and only if P=PP.

Stated more precisely,

$$\forall f \in \#\mathrm{P}, \exists c, \forall x f(x) \in K[c\log(|x|), |x|^c \mid x] \Longleftrightarrow \mathrm{P} = \mathrm{PP}.$$

Intuitively the result shows that the question of whether #P functions can generate randomness is equivalent to the P=PP question. The proof of this fact uses some powerful recent results in complexity theory, such as the functional version of Toda's theorem [12] and the interactive protocol for computing the permanent from [9].

Our result is also connected with a certain kind of approximation of #P functions [3]. A direct consequence of the fact $\mathrm{PH} \subseteq \mathrm{P}^{\mathrm{PP}}$ [11] is that #P functions cannot be computed using resources from the polynomial time hierarchy unless PH collapses. On the other hand

Stockmeyer [10] shows that the value of a #P function can be approximated within a multiplicative factor by a function in Δ_3^p. A probabilistic approximation of #P using Δ_2^p resources is obtained in [6].

Cai and Hemachandra study in [3] a new approach to approximating #P functions. They define the concept of an s-enumerator of a function f. Such an enumerator on input x produces a list of $s(|x|)$ possible values for $f(x)$ in which $f(x)$ appears. In their article it is shown that for any $\alpha < 1$, #P functions have polynomial time n^α-enumerators if and only if P=PP. However the authors leave the case of general polynomial size enumerators as an open problem. In fact the statement that #P functions have polynomial size enumerators which run in polynomial time is equivalent to the statement that #P functions have Kolmogorov simple images (see lemma 6 below). Hence from our result it follows that #P has polynomial size enumerators iff P=PP, thus solving the open question proposed in [3].

2. Preliminaries

The classes of languages of importance in this article are P, NP, PP and R. The reader is referred to [2] for precise definitions of these classes.

The class #P of functions that count the number of accepting paths of a nondeterministic Turing machine was defined by Valiant in [13]. He proved that the function *perm* which calculates the permanent of an integer matrix is complete for #P under a certain type of functional reduction. Recall that the permanent of an $n \times n$ matrix $A = (a_{i,j})$ is defined as

$$perm(A) = \sum_{\sigma} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

where σ is one of the possible permutations of $\{1, \ldots, n\}$.

For a complexity class $\mathcal{C}$ we denote by $\#\mathrm{P}^{\mathcal{C}}$ the class of functions which count the number of accepting path of a nondeterministic Turing machine with access to an oracle A in the class $\mathcal{C}$.

It is well known that polynomial-time computable functions (functions in FP) are in #P, and that #P is closed under sum and product. The following lemma follows from these properties.

Lemma 3 *Let $f_1, \ldots, f_n$ be a list of #P functions and let p be a polynomial bounding the size of all these functions (for every $x \in \Sigma^*, i \leq n$, $|f_i(n)| < p(|x|)$). The function $f(x) = 2^{(n-1)p(|x|)} f_1(x) + 2^{(n-2)p(|x|)} f_2(x) + \ldots + f_n(x)$ is in the class #P.*

This lemma can be consider as a way to encode the values of $f_1, \ldots, f_n$ in the value of f. Observe that we can obtain in polynomial time the values of $f_i(x)$ from the value of $f(x)$. Such an encoding of functions will be denoted in the paper by $\langle f_1, f_2 \ldots, f_n \rangle$. We will need in our proofs the following stronger version of lemma 3, which includes the concept of uniformity.

Lemma 4 *Let $f_1, f_2, \ldots, f_n, \ldots$ be a (possibly infinite) list of #P functions which compute the number of accepting paths of nondeterministic time machines $M_1, \ldots, M_n, \ldots$, such that there*

is a function $g \in FP$ which on input 0^i outputs the description of machine M_i. Let h be a function in FP of polynomialy bounded size. The function $f(x) = \langle f_1(x), f_2(x), \ldots, f_{h(|x|)}(x)\rangle$ is in the class $\#P$.

We will characterize functions with Kolmogorov simple images in a more practical way. For this we use the following definition from [3].

Definition 5 Let $f : \Sigma^\star \longrightarrow \mathbb{N}$, and $s : \mathbb{N} \longrightarrow \mathbb{N}$ be two functions. The function E is an s-enumerator for f if for every x, $E(x)$ is a list of at most $s(|x|)$ integers in which $f(x)$ appears.

Cai and Hemachandra show that #P functions have n^α-enumerators (for any $\alpha < 1$) which run in polynomial time iff P=PP. We will improve this result to the case of polynomial time p-enumerators where p is any polynomial. We first need the following lemma which makes the straightforward connection between p-enumerators and Kolmogorov simple images.

Lemma 6 *Let $f : \Sigma^\star \longrightarrow \mathbb{N}$. The following statements are equivalent.*

i) *f has Kolmogorov simple images relative to the input.*

ii) *There is a polynomial p such that f has a p-enumerator which runs in polynomial time.*

Proof: i)⇒ii). If f has Kolmogorov simple images relative to the input then for a certain constant c, for every x, $f(x) \in K[c\log(|x|), |x|^c \mid x]$. A polynomial time $2n^c$-enumerator for f, E, can be constructed in the following way: On input x, E simulates the universal Turing machine M_U on all possible strings $\langle i, x\rangle$, with $|i| \leq c\log(|x|)$, printing the output M_U on the list of possible values if the universal Turing machine halts within $|x|^c$ steps. One of the output values is the correct one by the hypothesis.

ii)⇒i). Suppose f has a polynomial time p-enumerator which runs in polynomial time. For sufficiently large c, it holds that for every x, $f(x) \in K[c\log(|x|), |x|^c \mid x]$ since $f(x)$ can be obtained from the program of the enumerator and the encoding of the position of $f(x)$ in the list produced by it. Observe that the size of the encoding of the position has logarithmic length because there is only a polynomial number of possible values. ∎

3. Main Result

In this section we will show that #P functions have Kolmogorov simple images relative to the input if and only P=PP. The proof from left to right is divided in two parts. Lemma 7 shows that the hypothesis implies P=NP; later we show that it also implies $PP \subseteq NP^R$. From these two results P=PP follows under the hypothesis mentioned.

Lemma 7 *If #P functions have Kolmogorov simple images relative to the input then P=NP.*

Proof: We will show that under the hypothesis, the set of satisfiable boolean formulas, SAT, is in P. The proof is based on the following result from [12]: for every k, $\#\mathrm{P}^{\Sigma_k^p} \subseteq \mathrm{FP}^{\#\mathrm{P}[1]}$, i.e, every function in $\#\mathrm{P}^{\Sigma_k^p}$ can be computed by a polynomial-time algorithm querying a #P function just once.

The function *max* which on input F outputs the maximum satisfying assignment of F if $F \in$ SAT and 0 otherwise, is in the class $\mathrm{FP}^{\mathrm{NP}}$ and therefore in $\#\mathrm{P}^{\mathrm{NP}}$. By the result from [12] there is a function $f \in \#\mathrm{P}$ such that $max \in \mathrm{FP}^{f[1]}$. Let M be the polynomial time machine with access to f which computes *max*. By the hypothesis of the lemma f has Kolmogorov simple images relative to the input. By lemma 6, for some polynomial p, f has a polynomial time p-enumerator E. For a sufficiently large polynomial q, one can construct a q-enumerator E' for *max* in the following way: On input F, simulate the polynomial time machine M which computes *max*. If the machine makes query $g(F)$ to the functional oracle f, run E on $g(F)$. Continue the computation of M on each of the $p(|g(F)|)$ possible values produced by E, producing the list of different outputs of the computations of M.

We have shown that *max* has a q-enumerator E' which runs in polynomial time. This implies SAT$\in$P since in order to decide whether a formula F is satisfiable, one can run the enumerator E' on F and check whether at least one of the produced values is a satisfying assignment of F. ∎

Lemma 8 *Let A and B be two integer matices of size $n \times n$, and consider the polynomial on variable x $perm(xA + (1-x)B)$. The function f which on input $\langle A, B, m\rangle$ outputs the coefficient of the term of degree m in $perm(xA + (1-x)B)$ is in the class $\#P$.*

Proof: Let $C(x) = (c_{i,j})$ be the $n \times n$ matrix $(xA + (1-x)B)$. $perm(C(x))$ is the polynomial on x

$$perm(C(x)) = \sum_{\sigma} \prod_{i=1}^{n} c_{i,\sigma(i)}$$

where σ is a permutation of the set $\{1, \ldots, n\}$. We describe a nondeterministic machine M which on input $\langle A, B, m\rangle$ produces a number of accepting paths equal to the value of the coefficient of degree m of $perm(C(x))$.

input $\langle A, B, m\rangle$;
guess a permutation of $\{1, \ldots, n\}$, σ;
compute k, the coefficient of x^m in the n^{th} degree polynomial in x, $\prod_{i=1}^{n} c_{i,\sigma(i)}$
guess a positive integer $l \leq k$ and accept.

For every permutation σ, M produces as many accepting paths as the value of the coefficient of x^m in that term of the sum which computes the permanent. The sum of the number of accepting paths over all possible permutations is therefore the coefficient of x^m in $perm(C(x))$. ∎

By lemma 4 there is also a #P function producing an encoding of the coefficients of every degree in the polynomial $perm(C(x))$. We use this fact extensively in the proof of the main theorem, which follows.

Theorem 9 #P *has polynomial-time p-enumerators if and only if* P = PP.

Proof: If P=PP then $FP^{PP} = FP = \#P$. Hence an FP-machine can produce the value of any #P function. A polynomial-time enumerator would work by simply outputting this single value.

For the converse, assume there exists a polynomial-time p-enumerator for #P. Recall that by Lemma 7 this implies that P=NP.

Let $L = \{\langle A, x\rangle | x = perm(A)\}$. Since the permanent is Turing-hard for #P [13], it is not hard to see that $PP \subseteq NP^L$. We describe a randomized algorithm for L. Later we will show that the algorithm has a co-R acceptance behavior. Therefore we find that $PP \subseteq NP^R$ and hence, since P=NP, that $PP \subseteq P$.

Before describing the algorithm formally, we give the intuition behind it. The algorithm follows very closely the interactive protocol of [9] for computing the permanent. Here the role of the prover is played by the polynomial-time p-enumerator. One crucial difference with the interactive protocol is that rather than a single value which a prover would send, an enumerator sends a list of possible values. However, while in the case of a prover it is possible that the value sent is incorrect, we always know that one of the values on the list sent by the enumerator is guaranteed to be correct. The idea of the algorithm is then to eliminate any elements on the list provided by the enumerator that are incorrect. We use exactly the same strategy as [9] to accomplish this. Thus although the algorithm is not an interactive protocol, it helps to think of it as though it were, with the prover replaced by the enumerator. By combining permanents into low-degree polynomials and generating random numbers in an appropriate finite field we can discover the elements of the list on which the enumerator "cheats" (i.e., by giving us the wrong values) with high probability. Along the way we find it necessary to keep essentially a list of the "histories" of the protocol with the enumerator, represented by the polynomial-length vectors of values as described below.

As in [9], in order to exploit the properties of low-degree polynomials we perform calculations modulo a prime r which is chosen larger than the possible value of the permanent. Since, by the hypothesis, P=NP, we can find an appropriate prime in polynomial time. Henceforth all calculations (i.e., the consistency checks we describe below) are performed modulo this prime, and when the algorithm calls for a random number, it will be chosen according to the uniform distribution over the finite field F_r of characteristic r. The prime will be of polynomial length, so for some c, the number of elements in F_r is at least 2^{n^c} (although it is also no larger than some exponential) where n is the size of the matrix A.

We will make use of a #P function f defined as follows. $f(\langle A\rangle)$ computes the permanent of A as well as the permanents of the minors of A (denoted $A_1^{(n)}, A_2^{(n)}$, ... $A_n^{(n)}$) to return a "vector" of values $\langle perm(A^{(n)}), perm(A_1^{(n)}), perm(A_2^{(n)}), ..., perm(A_n^{(n)})\rangle$. With additional parameters f combines the minors of A to compute coefficients as described in Lemma 8. For example, $f(\langle A, y\rangle)$ $(y \in F_r)$ produces a vector of permanents of A and its minors with an additional component giving the coefficients of the polynomial $perm((1-x)A_1^{(n)} + xA_2^{(n)})$. Additional components in the input which are matrices yield additional vectors of permanents; and additional y's in the input yield additional coefficients of polynomials, in a fashion which is clarified in the algorithm below. By the uniformity lemma 4, the function f is computable in #P.

A formal statement of the algorithm to recognize L is now given. Assume the input is $\langle A, x\rangle$. *Enum* denotes a polynomial enumerator for f. The algorithm has different stages numbered from n down to 0.

Stage n.

Let $A^{(n)}$ denote A to start with. Run *Enum* on $\langle A^{(n)}\rangle$ to obtain a list of vectors of the form $\langle perm'(A^{(n)}), perm'(A_1^{(n)}), perm'(A_2^{(n)}), ..., perm'(A_n^{(n)})\rangle$. Verify $perm'(A^{(n)}) = \sum_{i=1}^{n} a_{1,i} perm' A_i^{(n)}$ for each element of the list, and reject any list element for which this is not true. Let $B_1^{(n)} := A_1^{(n)}$ (subsequent $B_i^{(n)}$'s are defined in the algorithm). In the following, by definition, for each i between 1 and $n-1$, $p_i^{(n)}(x) = perm((1-x)B_i^{(n)} + xA_{i+1}^{(n)})$. Note for each i that $p_i^{(n)}(x)$ is a polynomial of degree n.

for $i := 1$ **to** $n-1$ **do**

- Run *Enum* on the vector $\langle A^{(n)}, y_1^{(n)}, ..., y_{i-1}^{(n)}\rangle$ (note when $i = 1$ there are no $y_i^{(n)}$-components), to obtain a list of vectors as previously: $\langle perm(A^{(n)}),\ perm(A_1^{(n)}),\ perm(A_2^{(n)}), ..., perm(A_n^{(n)}), p_1^{(n)}(x), ..., p_i^{(n)}(x)\rangle$ Note that although the list may be longer than the previous list it is still of polynomial length and the number of items on the list is bounded by a polynomial (this is explained more precisely at the end of the proof).
- Keep only those list elements whose first $n+i$ components are identical with corresponding components of some vector on the previous list.
- For each new list element check $p_i'^{(n)}(0) = perm'(A_i^{(n)})$ and $p_i'^{(n)}(1) = perm'(A_{i+1}^{(n)})$. Reject any list element for which either of these relations does not hold.
- Generate $y_i^{(n)} \in F_r$ at random and record $p_i'^{(n)}(y_i^{(n)})$ for each list element that survives the above consistency tests.
- Let $B_{i+1}^{(n)} := (1-y_i^{(n)})B_i^{(n)} + y_i^{(n)} A_{i+1}^{(n)}$.

end for-loop.

Let $inputs^{(n)} := \langle A^{(n)}, y_1^{(n)}, ..., y_{n-1}^{(n)}\rangle$, $A^{(n-1)} := B_n^{(n)}$ and go to stage $n-1$.
end of stage n.

Stage j.

Denote the minors of $A^{(j)}$ as $A_1^{(j)}, A_2^{(j)}, ...\ A_j^{(j)}$. Run *Enum* on the input $\langle inputs^{(j-1)}, A^{(j)}\rangle$ to obtain a list of vectors as in stage n, $\langle prefix'^{(j-1)}, perm'(A^{(j)}), perm'(A_1^{(j)}), perm'(A_2^{(j)}), ..., perm'(A_j^{(j)})\rangle$, where $prefix'^{j-1}$ is a set of components of the form, $\langle prefix'^{j-2}, perm'(A^{(j-1)}), perm'(A_1^{(j-1)}), perm'(A_2^{(j-1)}), ..., perm'(A_{j-1}^{(j-1)}), p_1^{(j-1)}(x), ..., p_i^{(j-1)}(x)\rangle$. The difference here is the presence of the prefixes $prefix'^{(j-1)}$. We keep only vectors whose prefixes are consistent with the corresponding components of the previous list of vectors. In fact this is the only difference with stage n; the remainder of stage j is identical to the remainder of stage n,

with the index n replaced by j and all inputs containing the prefix $inputs^{(j-1)}$ preceeding the other components specified in stage n.
end of stage j.

If we arrive at stage 1, we can perform the consistency checks directly on the resulting matrices of size 1. Finally, if we arrive at stage 0, accept iff x is equal to the first component ($perm'(A^{(n)})$) of any one of the vectors on the remaining list.

This concludes the description of the algorithm. We now prove that it works correctly. Note, as promised, that the following claim shows that the algorithm for L is a co-R algorithm.

Claim: If $\langle A, x\rangle \in L$ then all paths accept, and if $\langle A, x\rangle \notin L$, then with high probability any path will reject.

proof of claim: Suppose $\langle A, x\rangle \in L$, so that $x = perm(A)$. Since *Enum* always produces a correct vector, such a vector will pass all consistency tests, including consistency with some previous vector on the list, and it will therefore always have $perm(A)$ as its first component. Then in the final stage there exists a list element with $x = perm(A)$ in the first component.

On the other hand, suppose $\langle A, x\rangle \notin L$, so $x \neq perm(A)$. Thus $x = perm'(A^{(n)})$ where $perm'(A^{(n)})$ is not the correct value. The remainder of this argument holds for any vector containing this value in its first component. We will show that any such vector will eventually be rejected with high probability.

By our consistency checks, we must have for some i that $perm(A_i^{(n)}) \neq perm'(A_i^{(n)})$. We also check $p_{i-1}'^{(n)}(1) = perm'(A_i^{(n)})$, so $p_{i-1}'^{(n)}(1) \neq p_{i-1}^{(n)}(1)$ and therefore in general the polynomials $p_{i-1}'^{(n)}(x)$ and $p_{i-1}^{(n)}(x)$ are different. Now consider any vector of the form, $\langle perm'(A^{(n)}), perm'(A_1^{(n)}), ...perm'(A_n^{(n)}), p_1'^{(n)}(x), p_2'^{(n)}(x), ..., p_i'^{(n)}(x)\rangle$, generated in step i by the enumerator, which is consistent with a previously generated vector that has passed all consistency tests. We have the relations $p_{i-1}'^{(n)}(y_{i-1}^{(n)}) = perm'(B_i^{(n)})$ and $p_{i-1}^{(n)}(y_{i-1}^{(n)}) = perm(B_i^{(n)})$. Thus since $p_{i-1}'^{(n)}(x) \neq p_{i-1}^{(n)}(x)$, either $y_{i-1}^{(n)}$ is a root of the polynomial $p_{i-1}'^{(n)} - p_{i-1}^{(n)}$ or $perm'(B_i^{(n)}) \neq perm(B_i^{(n)})$. Since $p_{i-1}'^{(n)} - p_{i-1}^{(n)}$ is a polynomial of small degree (n) the probability that $y_{i-1}^{(n)}$ is a root is exponentially small, and we can conclude that the probability that $perm'(B_i^{(n)}) \neq perm(B_i^{(n)})$ is exponentially close to 1. More precisely, $\Pr(perm'(B_i^{(n)}) \neq perm(B_i^{(n)})) \geq 1 - 2^{n^c}$. (Recall that 2^{n^c} is the lower bound on the size of the finite field F_r.) This error propagates to subsequent steps (values of i), with high probability, as follows. If $perm'(B_i^{(n)}) \neq perm(B_i^{(n)})$ then $p_i'^{(n)}(0) \neq p_i^{(n)}(0)$. Therefore for general x, $p_i'^{(n)}(x) \neq p_i^{(n)}(x)$ which implies that either $p_i'^{(n)}(y_i^{(n)}) \neq p_i^{(n)}(y_i^{(n)})$ (which is equivalent to $perm'(B_{i+1}^{(n)}) \neq perm(B_{i+1}^{(n)})$) or $y_i^{(n)}$ is a root of $p_i'^{(n)} - p_i^{(n)}$. Therefore $\Pr(perm'(B_{i+1}^{(n)}) \neq perm(B_{i+1}^{(n)})) \geq \Pr(perm'(B_i^{(n)}) \neq perm(B_i^{(n)})) - 2^{n^c}$. Proceeding in this fashion we can prove by induction that for any future vector consistent with an incorrect one, we have (for some c' and for any j and i) that $\Pr(perm'(B_i^{(j)}) \neq perm(B_i^{(j)})) \geq 1 - 2^{n^{c'}}$.

When $j = 1$ we can check the error explicitly. Hence any errors in vectors sent by the enumerator in earlier stages will be detected later on with probability $1 - 2^{n^{c'}}$, and such vectors will be eliminated by stage 0 with this probability. This proves the claim. ∎

Finally we explain more precisely the remark in the "for-loop" of stage n that the length of the list produced by the enumerator is bounded by a polynomial. Every element of an input vector has length bounded by a polynomial, say $q(n)$. There fewer than $O(n^2)$ elements in any vector, so any vector has length at most $O(n^2 q(n))$. Thus an n^k-enumerator will never produce more than $O(n^{2k}(q(n))^k)$ vectors, which is still a polynomial. This concludes the proof of the theorem. ∎

The main result is now immediate by lemma 6.

Corollary 10 #P *has Kolmogorov-simple images relative to the input if and only if P=PP.*

The following is a corollary of the proof of theorem 9.

Corollary 11 *If* #P *has p-enumerators in* $\mathrm{FP}^{\Sigma_k^p}$ *then* $\mathrm{PP} \subseteq \Sigma_{k+2}^p$.

By lemma 6 it is therefore unlikely that #P has Kolmogorov-simple images relative to any Σ_k^p, since by Toda's theorem if $\mathrm{PP} \subseteq \Sigma_{k+2}^p$ then the polynomial hierarchy collapses.

4. Acknowledgements

F.G. would like to thank the Departament de L.S.I., Universitat Politècnica de Catalunya, Barcelona for its hospitality while this research was being done, with special thanks to José Balcázar for helping to make the visit possible.

References

[1] L. Adleman, "Time, space and randomness", Technical Report MIT/LCS/TM-131, MIT, Cambridge, MA, April 1979.

[2] J. Balcázar, J. Díaz, and J Gabarró, *Structural Complexity Theory*, Springer-Verlag 1989.

[3] J. -Y. Cai and L. A. Hemachandra, "Enumerative counting is hard", *Information and computation* **82** (1989) 34-44.

[4] J. Hartmanis, "Generalized Kolmogorov complexity and the structure of feasible computations", *Proceedings 24th IEEE symposium on foundations of computer science* (1983) 439-445.

[5] L. A. Hemachandra and G. Wechsung, "Using randomness to characterize the complexity of computation", *Proceedings IFIP conference* (1989) 281-286.

[6] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani, "Random generation of combinatorial structures from a uniform distribution", *Theoretical computer science* **43** (1986) 169-188.

[7] M. Krentel, "The complexity of optimization problems", *Journal of computer and system science*, **36** (1988) 490-509.

[8] M. Li and P. Vitanyi, "Applications of Kolmogorov complexity in the theory of computation", in *Complexity Theory Retrospective*, A. Selman, ed., Springer-Verlag (1990) 147-203.

[9] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "The polynomial-time hierarchy has interactive proofs", *Proceedings 31st IEEE symposium on foundations of computer science* 1990.

[10] L. Stockmeyer, "On approximation algorithms for #P", *SIAM J. Comput.* **14** (1985) 849-861.

[11] S. Toda, "On the computational power of PP and $\oplus$P", *Proceedings 30th IEEE Symposium on Foundations of Computer Science* (1989) 514-519.

[12] S. Toda and O. Watanabe, "Polynomial time 1-Turing reductions from #PH to #P", *Theoretical computer science*, to appear.

[13] L. Valiant, "The complexity of computing the permanent", *Theoretical computer science* **8** (1979) 189-201.