Luís Filipe Coelho Antunes

# Useful Information


**Departamento de Ciência de Computadores**
**Faculdade de Ciências da Universidade do Porto**

**2002**

Luís Filipe Coelho Antunes

# Useful Information


*Tese submetida à Faculdade de Ciências da*
*Universidade do Porto para obtenção do grau de Doutor*
*em Ciência de Computadores*

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

2002

# Acknowledgments

During my graduate studies I was very fortunate to meet and work with some really good researchers and friends. I have tried to learn as much as possible with each one of them and I want to thank them here.

Foremost I want to thank my advisor, Armando Matos. Without him I would have never worked on Computational Complexity. Since the beginning he is very enthusiastic and supportive with my work. I have learned a lot with his inviting attitude towards research and knowledge. His permanent good mood was always contagious.

Jointly with Armando, we have decided that we should try to find some collaboration with other researchers. I was interested in Kolmogorov Complexity and Computational Complexity and Lance Fortnow was/is one of the most active researchers in this subject, so I decided to email him asking if I could visit him in Chicago. Lance was very enthusiastic about it and since then he has collaborated on most of my research. Every problem I have ever looked I have discussed with him or was suggested by him. I am thankful to Lance by his advice and support, his knowledge and intuition where invaluable. With his inviting attitude towards research I have learned a lot.

I want to thank my co-authors, Lance, Dieter and Vinod the permission to use the material in the papers in this dissertation.

I have visited some institutes during my graduate studies. I am grateful to:
*(i)* CWI, working with Paul Vitányi was very exciting and fruitful. At CWI I have meet Ronald and Hein to both of them my thanks for their friendship.
*(ii)* DIMACS by allowing me to participate in the Special Year on Computational Intractability. There I meet Diter, I want to thank him his support and friendship.
*(iii)* NEC for all the support they gave during my visits to work with Lance.

I am grateful to LIACC and particularly the NCC group. Miguel Filgueiras has been very supportive, I want to thank his advice and his help with the French. I also want to thank Sabine for reading parts of this thesis. I want to thank also the other researchers in the group, particularly the soccer players (ricroc, lblopes, pmedas and rslopes) who always complain about my skills...

I am grateful to FEP by allowing me to do this work. In particular I want to thank all people in the mathematics group by their friendship.

For financial support I'm grateful to LIACC, NEC, Fundação Caloust Gulbenkian, Fundação Luso-Americana para o Desenvolvimento and specially to PRODEP III.

From a more personal view I want to thank all my friends for their friendship and support. I believe that the wealthy of a men is in the friends he have, and fortunately I consider myself wealthy. Due to space restrictions I'll name them by domains `ncc.up.pt`,`fep.up.pt`,`med.up.pt`,`math.uminho.pt`. In particular I have to thank to Sergio Melo and Manuela Aguiar.

I want to thank my brothers, sister and their families for their unconditional support. A tender and special gratitude to my *parents*, their support and encouragement where invaluable, every day that passes I admire them more.

Finally ***Cristina*** my wife, for her continuous *love*, support (even if I've took decisions that she would have preferred to see differently) and friendship ***I want to express here my*** gratitude and ***love for her***.

# Abstract

Kolmogorov complexity measures the amount of information in a string $x$ as the length of a shortest description of $x$. Random strings are incompressible, as the shortest description of this strings is the string itself. They are therefore deemed to contain a lot of information. However, random information may not be very useful from a computational point of view. In this thesis we introduce the notion of *Computational Depth* that measures the amount of nonrandom information in a string by considering the difference of two distinct Kolmogorov complexity measures. We consider three types of computational depth:

- Time-bounded computational depth, a clean notion capturing the spirit of Bennett's Logical Depth. We show that most strings have both high time-bounded computational depth as well as high logical depth. If one is interested in programs with running time bigger than exponential then one should use busy beaver computational depth, a measure which preserves the intuition of time-bounded computational depth while capturing all possible running times.

- Distinguishing computational depth, that measures the difference in difficulty between recognizing and generating a string. We show that for formulas with satisfying assignments of low depth, an assignment can be found efficiently.

- Shallow Sets, a generalization of sparse and random sets based on low depth properties of their characteristic sequences. We show that every recursive set that is reducible to a shallow set has polynomial-size circuits.

We prove that there is an interesting relation between computational depth and Levin's average case complexity. Li and Vitányi showed that when the inputs of an algorithm are distributed according to the universal distribution $\mathbf{m}$ the algorithm's

average case complexity of the algorithm is of the same order of magnitude of its worst case complexity. Since the universal distribution is not even recursive, this result fits poorly with the traditional average case complexity analysis, where the distribution must be "simple" (usually considering polynomial time computable or samplable distributions). Therefore, unless we can derive some kind of time bounded version of Li and Vitányi's result, the two subjects seem quite unrelated. We show that the running time $t$ of a machine is polynomial on average with respect to the time-bounded universal distribution if and only if $t$ is bounded by $2^{depth}$. This result can be viewed as a generalization of Li and Vitányi's result. Indeed, as $t$ goes unbounded, depth approaches 0 and and $\mathbf{m}^t$, the time bounded version of $\mathbf{m}$, approaches $\mathbf{m}$. As an interesting corollary we get that, if the running time $t$ of a machine is exponentially bounded in the depth, then $t$ is polynomial on average with respect to all computable distributions of certain time bound.

The information contained in an binary string is usually measured by its Kolmogorov complexity. We can divide that information into two parts: the part accounting for the useful regularity present in the object and the part accounting for the remaining accidental information. There can be several ways (model classes) in which the regularity is expressed. Kolmogorov has proposed the model class of finite sets, generalized later by Gács, Tromp and Vitányi to computable probability mass functions. The resulting theory is known as Algorithmic Statistics. The most general way to proceed is perhaps to express the regularity as a recursive function. The resulting measure has been called, by Koppel, the sophistication of the object. Koppel claimed that depth and sophistication for all infinite strings are equivalent. However the proof is wrong and uses a different definition of depth imposing totality in the functions defining depth. In this thesis we show that it is possible to have some discrepancy between depth and sophistication. Nevertheless, we consider the concept of sophistication as genuinely interesting and study its properties for finite strings. In particular we prove that there are strings $x$ of length $n$ with sophistication close to $n$. We introduce a new variant of sophistication, called Coarse Sophistication, and show that for all $x$ of length $n$, given $x$ and $O(\log n)$ bits, we can solve the halting problem for all programs $q$ of length smaller than $\frac{c-soph(x)}{2} - 2\log n$. Finally, we prove that coarse sophistication is equivalent to busy beaver computational depth.

# Resumo

A complexidade de Kolmogorov mede a quantidade de informação numa sequência $x$ como o comprimento da menor descrição de $x$. Sequências aleatórias são incompressíveis, visto que a menor descrição é a própria sequência. Logo, estas sequências contém muita informação. Contudo, sequências aleatórias não são muito úteis em complexidade computacional. Nesta tese introduzimos a noção de *Profundidade Computational* que mede a quantidade de informação não aleatória numa sequência considerando a diferença entre duas variantes da complexidade de Kolmogorov. Neste trabalho estudamos três tipos de profundidade computacional:

- *Profundidade computacional limitada pelo tempo*, uma noção simples que preserva a intuição da profundidade lógica de Bennett. Mostramos que a maioria das sequências têm elevada profundidade computacional limitada pelo tempo e elevada profundidade lógica. Se estivermos interessados em programas com tempo de execução superior a exponencial, devemos usar a profundidade computacional "busy beaver", uma medida que preserva a intuitição da profundidade computacional limitada pelo tempo, considerando todos os tempos de execução possíveis.

- *Profundidade computacional distinguível*, que mede a diferença entre a dificuldade de distinguir e a dificuldade de gerar uma sequência. Mostramos que para fórmulas com atribuições lógicas com baixa profundidade computacional, podemos encontrar uma atribuição de maneira eficiente.

- *Conjuntos pouco profundos*, uma generalização dos conjuntos esparsos e conjuntos aleatórios com base na baixa da profundidade das sequências características. Mostramos que todo o conjunto recursivo que seja redutível a um conjunto pouco profundo, possui circuitos de tamanho polinomial.

Provamos uma relação entre a complexidade computacional média de Levin e a profundidade computacional. Li e Vitányi mostraram que quando os dados de entrada de

um algoritmo seguem a distribuição universal $\mathbf{m}$, a complexidade média é da mesma ordem de grandeza da complexidade clássica. Como a distribuição universal não é recursiva, este resultado não é adequado para a análise da complexidade média, onde a distribuição deve ser "simples" (geralmente considerando distribuições computáveis em tempo polinomial ou amostragens polinomiais). Consequentemente, a menos que possamos provar uma versão do resultado de Li e do Vitányi limitada pelo tempo, os dois assuntos não estão relacionados. Mostramos que considerando $\mathbf{m}^t$, a distribuição universal limitada pelo tempo, o tempo de execução $t$ de uma máquina é polinomial em média se, e somente se $t$ é limitado por $2^{depth}$. Este resultado é uma generalização do resultado de Li e Vitányi. De facto, à medida que $t$ cresce, a profundidade computacional tende para 0 e $\mathbf{m}^{\mathbf{t}}$ tende para $\mathbf{m}$. Como corolário, se o tempo de execução $t$ de uma máquina for limitada exponencialmente pela profundidade, então $t$ é polinomial em média com respeito a todas as distribuições computáveis em determinado limite de tempo.

A informação contida numa sequência é medida geralmente pela complexidade de Kolmogorov. Podemos dividir essa informação em duas partes: a parte que mede a regularidade útil no objecto e a parte que mede a informação acidental restante. Existem diversas maneiras (classes) de exprimir a regularidade. Kolmogorov propôs como modelo conjuntos finitos, generalizado mais tarde por Gács *et al.* para funções de probabilidade computáveis. A teoria resultante é conhecida como "estatística algorítmica". A maneira mais geral de proseguir é defenindo a regularidade como uma função recursiva. A medida resultante foi chamada, por Koppel a sofisticação do objecto. Koppel tentou provar a equivalência entre a profundidade e a sofisticação para sequências infinitas. No entanto, a prova está errada e usa uma definição diferente de profundidade, impondo que as funções que a definem sejam totais. Nesta tese mostramos que é possível haver alguma discrepância entre a profundidade e a sofisticação. Não obstante, consideramos o conceito interessante e estudamos as suas propriedades para sequências finitas. Em particular provamos que há sequências $x \in \{0,1\}^n$ com sofisticação perto de $n$. Introduzimos uma nova variante da sofisticação, chamada sofisticação grosseira, e mostramos que para todo o $x \in \{0,1\}^n$, dado $x$ e $O(\log n)$ bits, podemos resolver o problema da paragem para todos os programas $q$ de comprimento menor do que $\frac{c-soph(x)}{2} - 2\log n$. Finalmente, provamos que a sofisticação grosseira é equivalente à profundidade computacional de "busy beaver".

# Résumé

La complexité de Kolmogorov mesure la quantité d'information dans une chaîne de caractères $x$ comme la longueur d'une description la plus courte de $x$. Des chaînes de caractères aléatoires sont incompressibles, car la description la plus courte d'une telle chaîne est elle-même. Elles sont donc considérées comme ayant beaucoup d'information. Cependant, l'information aléatoire peut ne pas être très utile d'un point de vue de calcul. Dans cette thèse nous présentons la notion de *Profondeur de Calcul* comme une mesure de la quantité d'information non aléatoire dans une chaîne de caractères en considérant la différence de deux mesures distinctes de complexité de Kolmogorov. Nous considérons trois types de profondeur de calcul:

- Profondeur de calcul bornée par le temps, une notion claire capturant l'esprit de la profondeur logique de Bennett. Nous prouvons que la plupart des chaînes de caractères ont en même temps des valeurs larges soit pour la profondeur de calcul bornée par le temps, soit pour la profondeur logique. Si on s'intéresse par les programmes avec temps de fonctionnement plus grand que l'exponentiel on devra utiliser la profondeur de calcul de "busy-beaver", une mesure qui préserve l'intuition de la profondeur de calcul bornée par le temps tout en capturant tous les temps de fonctionnement possibles.
- La profondeur de calcul de distinction, qui mesure la différence en difficulté entre identifier et produire une chaîne de caractères. Nous prouvons que pour des formules avec des attributions de satisfaction de basse profondeur, une affectation peut être trouvée efficacement.
- Ensembles peu profonds ("shallow sets"), une généralisation des ensembles creux et aléatoires basée sur des propriétés de basse profondeur de leurs ordres caractéristiques. Nous prouvons que chaque ensemble récursif qui est réductible à un ensemble peu profond a des circuits de taille polynômial.

Nous prouvons qu'il y a une relation intéressante entre la profondeur de calcul et la complexité de cas moyen de Levin. Li et Vitányi ont prouvé que quand les entrées

d'un algorithme sont distribuées selon la distribution universelle $\mathbf{m}$ la complexité de cas moyen de l'algorithme est du même ordre de grandeur de sa complexité de pire cas. En étant donné que la distribution universelle n'est pas récursive, ce résultat s'ajuste mal avec l'analyse traditionnelle de complexité de cas moyen, où la distribution doit être "simple" (habituellement on considère des distributions soit calculables, soit résultant d'une échantillonnage, en temps polynômial). En conséquence, à moins que nous puissions dériver un version bornée par le temps du résultat de Li et de Vitányi, les deux concepts semblent tout à fait indépendants. Nous prouvons que le temps de fonctionnement $t$ d'une machine est polynômial en moyenne dans la distribution universelle bornée par le temps si et seulement si $t$ est bornée par $2^{profondeur}$. Ce résultat peut être considéré comme une généralisation du résultat de Li et de Vitányi. En effet, si $t$ croît, la profondeur approche 0 et $\mathbf{m}^t$, la version bornée par le temps de $\mathbf{m}$, approche $\mathbf{m}$. Comme corollaire on obtient que, si le temps de fonctionnement $t$ d'une machine est exponentiellement borné par la profondeur, alors $t$ est polynômial en moyenne dans toutes les distributions calculables dans une certaine borne de temps. L'information contenue dans une chaîne de caractères binaire est habituellement mesurée par sa complexité de Kolmogorov. Nous pouvons diviser cette information en deux parties: celle qui concerne la régularité utile existant dans l'objet, et celle qui décrit l'information accidentelle restante. Il peut y avoir plusieurs façons (classes de modèles) pour exprimer la régularité. Kolmogorov a proposé la classe de modèles des ensembles finis, généralisée plus tard par Gács *et al.* aux fonctions calculables de probabilité de masse. La façon la plus générale de procéder est peut-être d'exprimer la régularité comme une fonction récursive. La mesure résultante est appelée, par Koppel, la sophistication de l'objet. Koppel a avancé que la profondeur et la sophistication sont équivalentes. Cependant la preuve est erronée et utilise une définition différente de profondeur en imposant la totalité de les fonctions qui donnent la profondeur. Nous prouvons qu'il est possible d'avoir des différences entre la profondeur et la sophistication. Cependant on considère la notion de sophistication intéressant et étudie ses proprietés. Nous prouvons qu'il y a des chaînes $x$ de longueur $n$ avec une sophistication proche de $n$. Nous introduisons une nouvelle variante de sophistication, appelée sophistication grossière, et nous prouvons que pour toutes $x$ de longueur $n$, donné $x$ et $O(\log n)$ chiffres binaires, on peux résoudre le problème de l'arrêt pour les programmes $q$ de longueur inférieur à $\frac{csoph(x)}{2} - 2\log n$. Finalement, nous prouvons que la sophistication grossière est équivalente à la profondeur de calcul de "busy-beaver".

# Contents

# 1

# Introduction

Kolmogorov [Kol65] introduced the concept of the complexity of a finite object $x$. Briefly Kolmogorov complexity ($K(x)$) measures the amount of information in a string $x$ as the length of a shortest description of $x$. This definition depends essentially on the method of decoding. However, Kolmogorov using the general theory of algorithms, made the critical observation that this definition is in a specific sense invariant, i.e., computer independent.

Randomly chosen strings have high Kolmogorov complexity, as the shortest description of this strings is the string itself. They are therefore deemed to contain a lot of information. However, random information may not be very useful from a computational point of view. So we need some method to measure the amount of nonrandom information in a string. To address exactly this problem has been the main subject of the work in this thesis.

We develop a notion of Computational Depth measuring the amount of nonrandom information in a string. The idea is simple: we consider the difference of two different Kolmogorov complexity measures. What remains is the nonrandom information we desire. Intuitively, computational depth measures the amount of nonrandom or ***useful information*** in a string. Formalizing this intuitive notion is tricky. A computationally deep string $x$ should take a lot of effort to be constructed from its shortest description. Incompressible strings are trivially constructible from their shortest description, and therefore computationally shallow.

We develop several types of computational depth based on this idea. We do not believe

that there is a single best type of computational depth. Rather different notions have different properties and applications that one can use as appropriate.

In this thesis we focus on three specific types of computation depth: Time-Bounded Computational Depth, Distinguishing Computational Depth and Shallow Sets.

### Time-Bounded Computational Depth

Time-bounded computational depth is the difference between Levin's $Kt$ complexity and the traditional unrestricted Kolmogorov complexity. Levin's $Kt$ complexity measures the "age" of a string, $Kt(x)$ is roughly the logarithm of the amount of time until $x$ is generated if one were to generate all strings from scratch.

Time-bounded computational depth with its simple definition captures the intuition behind Bennett's [Ben88] rather technical notion of Logical Depth. A string $x$ has large logical depth if it has short programs but these programs require considerable computation time. Computational depth has a similar property. We show that exponentially many strings of a given length have both large logical depth as well as large time-bounded computational depth.

The main drawback of the concept of time-bounded computational depth is the fact that it is only suitable for strings whose programs run in time at most exponential in the length of the string. This motivates the introduction of the concept of Busy Beaver Computational Depth, which preserves the intuition of basic computational depth while capturing all possible running times.

The intuition behind busy beaver computational depth is that, instead of considering a significance level as Bennett did, we incorporate this in the formula. Another important issue in this measure is that, instead of using the running time $t$, we use the inverse busy beaver of the running time, which means that we are scaling down this measure to program length.

We can divide the information contained in an object into two parts: the information accounting for the useful regularity present in the object and the information accounting for the remaining accidental information. Koppel [Kop87, Kop88, KA91] suggested to express the useful information as a recursive function. The resulting measure has been called the "sophistication" of the object. Although some of the

results published in [Kop87] are wrong, we consider the concept of sophistication as genuinely interesting. We show that there exist strings $x \in \Sigma^n$ with sophistication close to $n$. We discuss the stability of sophistication and introduce a new variant of sophistication called the Coarse Sophistication.

Furthermore, we prove the following result regarding coarse sophistication: given $x \in \Sigma^n$ and $O(\log n)$ bits, we can solve the halting problem for all programs $q$ of size smaller than $\frac{\text{csoph}(x)}{2} - 2\log n$. This is not only a very strong result, but it is also suggests a strong and important connection between sophistication and computational depth. In fact, by the slow growth law, the halting sequence is very deep (it can speed up any slow computation) and thus strings with high sophistication must be deep. Finally, we prove the equivalence between busy beaver computational depth and coarse sophistication.

**Time-Bounded Computational Depth vs Average Case Complexity**

The complexity of a problem is usually defined in terms of the worst case complexity of algorithms. However despite having a bad worst case behavior, many algorithms are frequently used in practice because they are efficient on average. It seems that the instances which cause the bad worst case complexity are rare in many practical applications. Thus, in some cases, the average case complexity of a problem is a more significant measure than its worst case complexity. We show an interesting relation between the computational depth of a string $x$, defined as the difference between $K^t(x)$ and $K(x)$, and Levin's average case complexity.

Li and Vitányi [LV97a] studied the universal distribution $\mathbf{m}$, introduced by Solomonoff [Sol64], which assigns $\frac{1}{2^{-K(x)}}$ weight to string $x$. They showed that, when the inputs to any algorithm are distributed according to the universal distribution, the average case complexity of the algorithm is of the same order of magnitude as its worst case complexity. Since the universal distribution is not even recursive, this result fits poorly with the traditional average case complexity analysis, where the distribution must be "simple" (usually considering polynomial time computable or samplable distributions). Therefore, unless we can derive some kind of time-bounded version of Li and Vitányi's result, the two subjects seem quite unrelated. We have accomplished exactly this, we proved that the running time $t$ of a machine is polynomial on average with respect to the time-bounded universal distribution if and only if $t$ is bounded by $2^{depth(x)}\text{poly}(|x|)$.

This can be viewed as a generalization of Li and Vitányi's result. Indeed, as $t$ goes unbounded, depth approaches 0 and $\mathbf{m}^t$, the time bounded version of $\mathbf{m}$, approaches $\mathbf{m}$. As an interesting corollary, we get that, if the running time $t$ of a machine is exponentially bounded in the depth, then $t$ is polynomial on average with respect to all computable distributions of a certain time bound.

### Distinguishing Computational Depth

Distinguishing computational depth is defined as the difference between polynomial-time bounded distinguishing complexity as developed by Sipser [Sip83] and polynomial-time bounded Kolmogorov complexity. This measures the difference between recognizing a string and producing it. Fortnow and Kummer [FK96] showed that under reasonable assumptions, there exist strings with high distinguishing computational depth.

We show that, if all the satisfying assignments of a formula $\phi$ have low distinguishing computational depth relative to $\phi$, then we can find an assignment in probabilistic quasi-polynomial time. We also show that an honest injective polynomial-time computable function cannot map strings of low depth to strings of high depth.

### Shallow Sets

Karp and Lipton [KL80] showed that the polynomial-time hierarchy collapses if $NP$ reduces to a sparse set. Bennett and Gill [BG81] showed that the polynomial-time hierarchy collapses if $NP$ reduces to a random set. Are these two separate results or just two specific examples of some more general principle? We show that the latter is true.

We introduce the notion of *Shallow Sets* to answer this question. Both sparse as well as random sets do not contain much information about $NP$ problems like satisfiability or about any other language. We can always simulate the effects of a random oracle by flipping coins. Shallow sets are sets where the initial segments of their characteristic sequence have similar poly-logarithmic time-bounded Kolmogorov complexity and traditional Kolmogorov complexity. Sparse sets and random sets are shallow.

Using Nisan-Wigderson generators [NW94], we show that, if a recursive set $A$ is

polynomial-time Turing reducible to a shallow set, then $A$ has polynomial-size circuits. This implies that the polynomial-time hierarchy collapses if any $NP$-complete set reduces to a shallow set. This generalizes both the result of Karp and Lipton [KL80] for sparse sets as well as the result of Bennett and Gill [BG81] on random sets.

### Thesis overview

In Chapter 2 we introduce all the necessary notions. In Chapter 3 we explain the notion of Computational Depth and study in detail three instantiations of this notion. This work was done in collaboration with Lance Fortnow and Dieter van Melkebeek. It has appeared in the proceedings of the Conference on Computational Complexity 2001 [AFvM01].

In Chapter 4 we explain the known results on sophistication, prove some new results and introduce coarse sophistication. We relate coarse sophistication with the halting problem and prove the equivalence between coarse sophistication and a instantiation of computational depth. This work was done in collaboration with Lance Fortnow during a summer internship at NEC 2001. It has not yet been published.

In Chapter 5 we explain average case computational complexity and generalize a result from Li and Vitányi on the use of the universal distribution in average case complexity. This work was done in collaboration with Lance Fortnow and V. Vinodchandran, and has appeared in a NEC technical report [AFV01].

# 2

# Preliminaries

In this chapter, we introduce basic concepts from computational complexity, Kolmogorov complexity as well as terminology used in the rest of the thesis. For a detailed study of Kolmogorov complexity we refer the reader to [LV97b]. Kolmogorov complexity measures the amount of information needed to describe a single object (usually a string). The Kolmogorov complexity of a string is the size of the smallest program producing the string. In computational complexity, we do not discuss the complexity of a single string. Instead, we measure the computational resources necessary to recognize or produce a set of strings. Computational complexity is the study of the inherent difficulty of computationally solving problems.

## 2.1 Notation and Computational Models

This section is based on the books [BDG95, Sip97, Pap85, LV97b] and on the preliminaries of the following PhD thesis [For89, vM99]. We give some definitions to establish notation.

**Definition 2.1.1**

1. *An* alphabet *is a non-empty, finite set. We shall use upper case Greek letters to denote alphabets. The cardinality of alphabet* $\Sigma$ *is denoted* $|\Sigma|$.

2. *A* symbol *is an element of an alphabet.*

3. *Given an alphabet $\Sigma$, a string or word over $\Sigma$ is a finite sequence of symbols from $\Sigma$. The* length *of a string $x$, denote $|x|$, is the number of symbols in $x$. $\epsilon$ is the empty string.*

4. *Given two strings $x$ and $y$ over $\Sigma$ the concatenation of $x$ and $y$, denoted $xy$, is the string $z$ which consists of the symbols of $x$, followed by the symbols of $y$.*

5. *$\Sigma^n$ is the set of all strings over $\Sigma$ with length $n$. The set of all strings over $\Sigma$ is denoted $\Sigma^*$.*

6. *A binary string $x$ is a* proper prefix *of a binary string $y$ if we can write $y = xz$ for $z \neq \epsilon$. A set $\{x, y, \ldots\} \subseteq \{0,1\}^*$ is* prefix-free *if for any pair of distinct elements in the set neither is a proper prefix of the other.*

7. *Given an alphabet $\Sigma$, a* language *over $\Sigma$ is a subset of $\Sigma^*$.*

The emphasis in binary sequences is only for convenience. Other finite objects can be encoded into strings in natural ways. Let $x, y, z \in \mathbf{N}$. Identify $\mathbf{N}$ and $\{0,1\}^*$ according to the correspondence

$$(0, \epsilon), (1, 0), (2, 1), (3, 00), (4, 01), \ldots.$$

Let $\langle \cdot, \cdot \rangle$ be a standard one-one mapping from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$, for technical reasons chosen such that $|\langle x, y \rangle| = |y| + |x| + 2\log|x| + 1$, for example $\langle x, y \rangle = x'y = 1^{\log|x|}0|x|xy$. This can be iterated to $\langle \langle \cdot, \cdot \rangle, \cdot \rangle$.

Each binary string $x = x_1 x_2 \ldots x_n$ has a special type of prefix code, called a *self-delimiting code*,

$$\bar{x} = 1^n 0 x_1 x_2 \ldots x_n.$$

This code is self-delimiting because we can determine where the code word $\bar{x}$ ends by reading it from left to right without backing up. Using this code we define the standard self-delimiting code for $x$ to be $x' = \overline{|x|}x$. It is easy to check that $|\bar{x}| = 2n+1$ and $|x'| = n + 2\log n + 1$.

**Definition 2.1.2** *A function $f : \Sigma^* \to \Sigma^*$ is* honest *if there exists a $k$ such that for every $x \in \Sigma^*$,*

$$|f(x)| \leq |x|^k + k \text{ and } |x| \leq |f(x)|^k + k$$

We also denote injective functions as *one-to-one* and surjective as *onto* functions.

**Definition 2.1.3** *The characteristic sequence of a set $L$ is:*

$$\chi_L(x) = \begin{cases} 1 & \text{if} \quad x \in L \\ 0 & \text{if} \quad x \notin L \end{cases}$$

**Definition 2.1.4** *A language is* tally *if and only if it is included in $\{a\}^*$ for some symbol $a$.*

**Definition 2.1.5** *The* census function *of a set $A$ over $\Sigma$, $C_A(n)$, is defined as the number of strings in $A$ having length at most $n$.*

**Definition 2.1.6** *A set $A$ is* sparse *if and only if its census function is bounded above by some polynomial, i.e., if there exists a polynomial $p(n)$ such that, for every $n$, $C_A(n) \leq p(n)$.*

**Lemma 2.1.7** *Every tally set is sparse.*

**Proof.** For any symbol $a$, every set $A$ included in $\{a\}^*$ is sparse, since for each $n$, there is at most one string with length equal to $n$ in $A$. Therefore the total number of strings of length less or equal to $n$ is at most $n + 1$. ◇

### 2.1.1 Turing machine

The most used model of computation is the Turing Machine.

**Model:** A *Turing machine*, consists of a finite state control connected to an input tape and work tape (see Figure 2.1) and a read write head for each of them. The finite control consists of a set of states including a specified initial state and accepting state, a transition function $\delta$ and the tape heads. Each head is located on one of the cells of its tape and can be moved right or left. The transition function $\delta$ take the values under the head of the input and work tape, the current state and describes whether to move the head right or left and switches to a new state. The transition function also may specify a change in the contents of the work tape cell under the head.

Before the Turing machine starts computing, an input string $x$ is placed on the input tape one symbol in each cell. The tape heads are positioned on the first cell of each tape. The Turing machine computes via the transition function $\delta$ in each step moving the head possibly changing the value of the cell in the work tape. The Turing machine accepts if it ever enters the accepting state. We say that the Turing machine accepts a language $L$ if it accepts as input strings exactly those strings in $L$. We let $L(M)$ designate the language accepted by the Turing machine $M$.

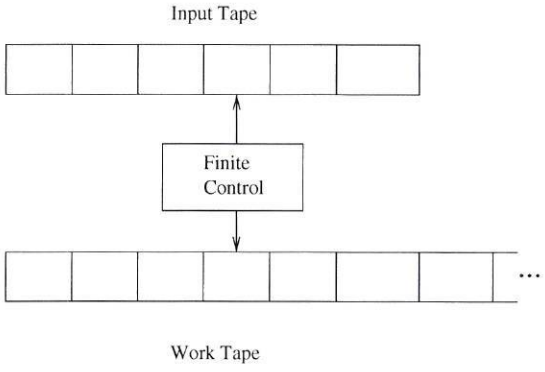


Figure 2.1: A Turing Machine.

We shall not only deal with the decision and acceptance of languages, but also with the computation of string functions. We add to the Turing machine model a write only output tape which initially is blank. When the Turing machine wishes to output a character, the output tape head writes the character and moves one space right. A Turing machine outputs a string $x$ on input $p$ if that machine outputs exactly the characters in $x$ before it halts.

An *instantaneous description* of the Turing machine contains all the information about its current configuration except for the input and output: the state of the finite control, the contents of the work tape, and the positions of the input tape and work tape heads. Note that at any time, only a finite portion of the tape is non-blank, so that each instantaneous description is finite. The sequence of instantaneous descriptions from the initial one forms a transcript of the computation of the given input.

A function that can be computed as described above by a Turing machine is called *partial recursive* or *computable*. If the Turing machine halts for all inputs, then the function computed is defined for all arguments and we call it *total recursive*, or simply *recursive*.

**Church's Thesis:** *The class of algorithmically computable numerical functions coincides with the class of partial recursive functions.*

Alternative definitions of Turing machines exists, including nondeterministic versions and versions with multiple tapes. Any model of computation equivalent to the Turing machine model is called *universal model.*

It is possible to give an effective one-to-one (injective) pairing between natural numbers and Turing machines. One way to do this is to encode the transition function of each Turing machine in binary, in a canonical way. This is called an *effective enumeration.* The effective enumeration of Turing machines $T_1, T_2, ...$ determines an effective enumeration of partial recursive function $\phi_1, \phi_2, ...$ such that $\phi_i$ is the function computed by $T_i$ for all $i$. An important distinction should be made between a function $\psi$ and a name for $\psi$. A name for $\psi$ can be an algorithm that computes $\psi$, such as a Turing machine $T$. It can also be a natural number $i$ (index for $\psi$), such that $\psi$ equal $\phi_i$ in the above enumeration. Thus, each partial recursive $\psi$ occurs many times in the given effective enumeration, i.e., it has many indices.

A *universal* Turing machine $U$ is a Turing machine that can simulate the behavior of any other Turing machine $T$. The machine $U$ receives as input the encoding of a machine $T$ together with an input $p$ to $T$, and simulates the computation of $T$ on $p$. These machines are similar to "interpreters" for computer programming languages, i.e., programs that read and execute other programs. The most fundamental result is that these machines do exist. In fact, there are infinitely many such $U$'s, and can be effectively constructed.

The existence of universal Turing machines lead us quickly to *undecidable* problems,i.e., problems that have no algorithm, languages that are not recursive. Once we accept the correspondence of problems with languages and algorithms with Turing machines, there are trivial reasons why there must be undecidable problems: there are more languages (uncountably many) than ways of deciding them (Turing machines).

**Definition 2.1.8 (Halting Problem)** *Given the description of a Turing machine $M$ and its input $x$, will $M$ halt on $x$?*

The following lemma, proved by diagonalization, formalizes this discussion. Let $\phi_1, \phi_2, ...$ be the standard enumeration of partial recursive functions.

**Lemma 2.1.9 (Turing)** *There is no recursive function $g$ such that for all $x, y$, we have $g(x, y) = 1$ if $\phi_x(y)$ is defined and $g(x, y) = 0$ otherwise.*

Define the *halting set* $K = \{\langle x, y\rangle : \phi_x(y) \text{ halts}\}$, we will use the notation $\phi(x) \downarrow$ ($M(x) \downarrow$) if $\phi(x)$ ($M(x)$) halts or converges and $\phi(x) \uparrow$ ($M(x) \uparrow$) otherwise; then Lemma 2.1.9 can be rephrased as: $K$ is not recursive. Note that the halting set defines a language that contains all strings that encode a Turing machine and an input, such that the Turing machine halts on that input.

**Lemma 2.1.10** *The halting set is recursively enumerable.*

To prove this lemma, we only need to change the universal Turing machine so that, whenever it halts, it does so in a "yes" state.
The halting set has also the following important property: if we had an algorithm deciding $K$, then we would have an algorithm deciding *all* recursively enumerable language. Let $L$ be an arbitrary recursively enumerable language accepted by a Turing machine $M$. Given $x$ as input, we would be able to decide whether $x \in L$ by simply telling whether $\langle M, x\rangle \in K$. We can then say that all recursively enumerable languages can be reduced to $K$, and that $K$ is complete for the class of all recursively enumerable languages. These concepts of reduction and completeness will be study in next section.

### 2.1.2 Non-deterministic Turing machines

In the description of a Turing machine every move is completely determined by the current situation. The state of the machine, and the symbols currently scanned by the tape heads, completely determine the next state and the moves of the tape heads. If we relax this condition we obtain nondeterministic devices, whose next move can be chosen from several possibilities.

Nondeterministic computation allows the Turing machine to make guesses. If a series of guesses leads to an accepting state then the Turing machine accepts. Formally, we let the transition function $\delta$ have a set of possible moves for a given state. We say the nondeterministic Turing machine $M$ accepts an input string $x$ if there is a choice of transitions that cause $M$ to enter an accepting state.

If the use of resources (time or space) is not considered, then deterministic machines have the same power as nondeterministic ones with an exponential loss of efficiency. So nondeterministic Turing machines can be used as universal models of computation.

**Theorem 2.1.11** *Every nondeterministic Turing machine has an equivalent deterministic Turing machine.*

**Proof.** (*Sketch*) Given a nondeterministic Turing machine $N$, design a deterministic Turing machine $M$ that on an input $x$ builds the computation tree of $N$ on input $x$ and performs a standard tree-search algorithm that halts and accepts if and only if it finds a leaf that is accepting.
The search should be done as a breadth-first, i.e., the computation tree is searched one level at a time, so if there is a leaf that is accepting, the simulation will find it. This way we avoid infinite computation paths. ◇

However, if we limit the resources then this may not hold, i.e., whether the exponential loss is inherent or an artifact of our limited understanding of nondeterminism is the famous $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ problem.

Unlike deterministic Turing machines, a nondeterministic Turing machine does not represent a real computing device. Rather, one should see nondeterministic Turing machines as a useful device for describing languages, and classifying computational problems.

### 2.1.3 Probabilistic Turing machines

The probabilistic model of computation is similar to the nondeterministic model. The main difference is that instead of "guessing" the next move, we "toss a coin" and make the move as a function of the outcome. The notion of acceptance is also changed: while in nondeterminism the input is accepted if and only if there is at least one computation that finishes in an accepting state, in the probabilistic machines we consider the probability of getting an accepting computation.

**Model:** we define *probabilistic Turing machine* by giving the Turing machine the ability to flip coins by allowing it access to an additional tape that contains the outcomes of independent fair coin tosses (see Figure 2.2). The tape is read-only and

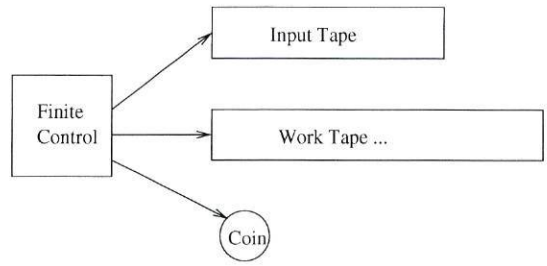


Figure 2.2: A Probabilistic Turing Machine.

the tape head can only move to the right. Reading a symbol from this tape and moving the tape head corresponds to flipping a coin.

When a probabilistic Turing machine recognizes a language, it must accept all strings in the language and reject all strings out of the language as usual, except that now we allow the machine a small probability of error. For $0 \leq \epsilon \leq \frac{1}{2}$ we say that a Turing machine $M$ recognizes a language $L$ with error probability $\epsilon$ if

1. $x \in L$ implies $Pr[M \text{ accepts } x] \geq 1 - \epsilon$, and
2. $x \notin L$ implies $Pr[M \text{ rejects } x] \geq 1 - \epsilon$

We also consider error probability bounds that depend on the input length $n$. For example, error probability $\epsilon = 2^{-n}$ indicates an exponentially small probability of error.

### 2.1.4 Relativized Turing machines

One can also modify the Turing machine model of computation by giving to it certain information essentially for "free". Depending on which information is actually provided (oracle), the Turing machine may be able to solve some problems more easily than before. An oracle $A$ is a language.

**Model:** a *relativized* Turing machine is a machine with a special oracle tape on which it writes a string $x \in \Sigma^*$ (see figure 2.3). The Turing machine then enters an *oracle query* state that immediately goes to a special *yes* state if $x \in A$ and to a *no* state otherwise. We only charge the Turing machine the time it requires to write down the oracle query. Nondeterministic and probabilistic Turing machines can be relativized in

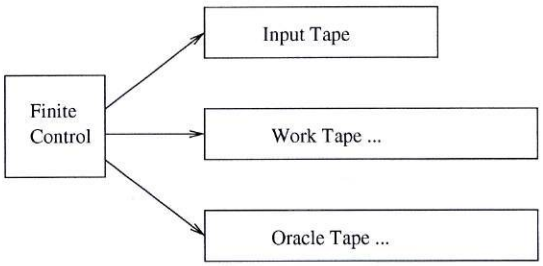


Figure 2.3: A Relativized Turing Machine

the same way. Superscripts will be used to represent access to an oracle. For example, $M^A$ represents a relativized Turing machine $M$ with access to oracle $A$.
Equivalently, we can think of an oracle as its characteristic function.

### 2.1.5 Circuits

Instead of computing via a machine, we can also compute via circuits.

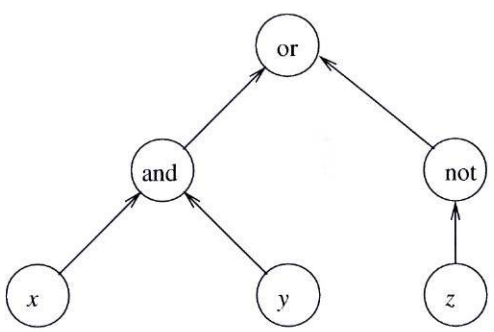


Figure 2.4: A Circuit

**Model:** A *Boolean circuit* is a combination of *not, or* and *and* gates as nodes of an acyclic directed graph with the input at the leaves. Each of the input gates is labeled with a variable $x_i$ for some $i \in \mathbf{N}$. Given a truth-value assignment $\sigma : \{x_i\}_i \to \{TRUE, FALSE\}$, we can inductively define a truth-value for each of the gates. The value of an input gate with label $x_i$ is $\sigma(x_i)$, and each of the other gates gets the result of applying the corresponding Boolean function to the values of the incoming gates. The circuit accepts if the value of the root node is 1.
The *size of the circuit* equals the number of gates it contains. The depth of the circuit is the length of the longest path. The *fan-in* of a gate $g$ is the number of gates pointing

to $g$.

A family of circuits $\mathcal{C} = \{C_1, C_2, ...\}$ consists of a set of circuits where $C_n$ has $n$ inputs. Suppose $x = x_1x_2...x_n$ has length $n$. Then $x$ is accepted by $\mathcal{C}$ if $C_n$ on $x_1x_2...x_n$ accepts. A family of circuits $\mathcal{C}$ has size (depth) $f(n)$ if $C_n$ has size (depth) at most $f(n)$ for all $n$.
Besides some size or depth restriction, circuits of the same family do not need to be related. Often we refer to such circuit families as *nonuniform* circuits.

## 2.2 Computational Complexity

This section is based on the books [BDG95, Sip97, Pap85] and on the preliminaries of the following PhD thesis [For89, vM99].
In order to impose bounds on the running time or on the working space of Turing machines, we must formally define these concepts. We stress that we will be mainly interested in the running time.
Usually one considers infinite computation problems, and are mainly interested in how the resources needed increase as the input size grows. Because the exact running time is a complex expression we usually just estimate it. The following notation allow us to compare the asymptotic growth rate of resource bounds up to constant factors. Let $f, g : \mathbf{N} \to [0, \infty)$ be functions.

- $f \in O(g)$ if the exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all sufficiently large $n \in \mathbf{N}$ ($f$ does not grows faster than $g$).

- $f \in \Omega(g)$ if the exists a constant $c > 0$ such that $f(n) \geq c \cdot g(n)$ for all sufficiently large $n \in \mathbf{N}$ ($f$ grows at least as fast as $g$).

- $f \in \Theta(g)$ if both $f \in O(g)$ and $f \in \Omega(g)$ ($f$ and $g$ have the same growth rate).

- $f \in o(g)$ if for any constant $c > 0$ and for large enough $n \in \mathbf{N}$, depending on $c$, $f(n) \leq c \cdot g(n)$ ($f$ grows slower than $g$).

- $f \in \omega(g)$ if for any constant $c > 0$ and for large enough $n \in \mathbf{N}$, depending on $c$, $f(n) \geq c \cdot g(n)$ ($f$ grows faster than $g$).

We will often derive bounds of the form: $O(\log n)$ called logarithmic, $n^{O(1)}$ called polynomial, $2^{n^{o(1)}}$ called sub-exponential and $2^{n^{O(1)}}$ called exponential.

**Definition 2.2.1** *The* running time *(*computation space*) of a Turing machine $M$, is the number of applications of the transition function (cells scanned on the work tape) until the machine reaches the final state. The Turing machine accepts a language $L$ in $t(n)$ steps if for all $x$ in $L$, $M$ accepts $x$ in at most $t(n)$ steps.*

In order to bound the running time and computation space we will typically use *constructible functions.*

**Definition 2.2.2** *A function $t$ $(s) : \mathbf{N} \to [0, \infty)$ is* time-constructible *(*space-constructible*) if there exists a Turing machine that runs in time (space) exactly $t(n)$ $(s(n))$ on every input of size $n$.*

All explicit resource bounds we use in this thesis are time-constructible.

### Deterministic time and space complexity

The complexity class DTIME$[f(n)]$ is the set of languages that can be decided in time $O(t(n))$ on a Turing machine. More time allows us to solve more problems.

**Definition 2.2.3** (P) P *is the class of all languages accepted in polynomial time, i.e.,*

$$\mathrm{P} = \cup_{k>0}\mathrm{DTIME}[n^k].$$

Another commonly used time-bounded class is exponential time

$$\mathrm{EXP} = \cup_{k>0}\mathrm{DTIME}[2^{n^k}].$$

We will also consider the class $\mathrm{E} = \cup_{k>0}\mathrm{DTIME}[2^{kn}]$.
In a similar way, we define the complexity class DSPACE$[f(n)]$. The complexity class PSPACE contains all the languages accepted in polynomial space. The class L contains the languages accepted in logarithmic space, DSPACE$[\log n]$.

**Theorem 2.2.4 (Time Hierarchy Theorem [HS65])** *Let $t_1, t_2 : \mathbf{N} \to [0, \infty)$ be such that $t_2$ is time-constructible, and $t_2 \in \omega(t_1 \log t_1)$. Then* $\mathrm{DTIME}[t_1] \subsetneq \mathrm{DTIME}[t_2]$.

The time hierarchy theorem implies that

$$\mathrm{P} \subsetneq \mathrm{E} \subsetneq \mathrm{EXP}.$$

Hartmanis *et al.* [SHI65] proved a similar space hierarchy theorem analog that implies that

$$\mathrm{L} \subsetneq \mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE}.$$

### Nondeterministic time and space complexity

As in the case of deterministic Turing machines, we define the complexity class NTIME$[f(n)]$ as the set of languages that can be decided by a nondeterministic Turing machine in time $O(t(n))$. For a given $x$, the running time (space) is defined as the maximum over all possible computation paths on input $x$ of the time (space) needed along that path. More time allows us to solve more problems. The complexity class NP contains all the languages with short efficiently verifiable membership proofs.

**Definition 2.2.5** (NP) NP *is the class of all languages with short efficiently verifiable membership proofs, i.e.,*

$$\mathrm{NP} = \cup_{k>0}\mathrm{NTIME}[n^k].$$

We will use the class

$$\mathrm{NEXP} = \cup_{k>0}\mathrm{NTIME}[2^{n^k}]$$

, and also $\mathrm{NE} = \cup_{k>0}\mathrm{NTIME}[2^{kn}]$. The classes NL and NPSPACE are the nondeterministic analogs of L and PSPACE. As a consequence of Savitch's theorem [Sav70], which shows how to deterministically simulate nondeterministic computations with a quadratic overhead in space, PSPACE = NPSPACE.

**Theorem 2.2.6 (Nondeterministic Time Hierarchy Theorem [Ž83])** *Let* $t_1, t_2 : \mathbf{N} \to [0, \infty)$ *be such that* $t_2$ *is time-constructible, and* $t_2(n) \in \omega(t_1(n+1))$. *Then* $\mathrm{NTIME}[t_1] \subsetneq \mathrm{NTIME}[t_2]$.

It follows from the nondeterministic time hierarchy theorem that

$$\mathrm{NP} \subsetneq \mathrm{NE} \subsetneq \mathrm{NEXP}.$$

Clearly NTIME[$t(n)$] contains DTIME[$t(n)$]; however we do not know if the converse is true. The best we know is that NTIME[$t(n)$] $\subseteq \cup_{k>0}$DTIME[$2^{k \cdot t(n)}$]. In particular, the smallest deterministic time-bounded class known to contain NP is EXP.

One of the greatest unsolved problems in theoretical computer science is whether P = NP. If these classes where equal, any polynomially verifiable problem would be polynomially decidable. Most researchers conjecture that P $\neq$ NP, i.e., coming up with a proof is harder than checking one.

For a complexity class $\mathcal{C}$, we let co-$\mathcal{C}$ contain all languages whose complement belongs to $\mathcal{C}$. For any two complexity classes, $\mathcal{C}$ and $\mathcal{D}$, if $\mathcal{C} \subseteq \mathcal{D}$ then co-$\mathcal{D} \subseteq$ co-$\mathcal{C}$.

A important question for nondeterministic complexity classes is whether they are closed under complementation. One can prove that co-NTIME[$t(n)$] $\subseteq \cup_{k>0}$NTIME[$2^{k \cdot t(n)}$] by exhaustively ruling out all possible proofs of length $O(t(n))$. However, this is the best we know. It is usually conjectured that polynomial size proofs for coNP languages do not exist, i.e., that

$$\text{NP} \neq \text{coNP}.$$

### Reductions and Completeness

A *reduction* is a way of converting one problem into another problem in such a way that a solution to the second problem can be used to solve the first problem. Formally, a reduction of a problem $A$ to a problem $B$, written $A \leq B$, is an oracle program for $A$ with $B$ as the oracle. Note that the reducibility says nothing about solving $A$ or $B$ alone, but only about the solvability of $A$ in the presence of a solution to $B$. Various types of reductions can be distinguished based on the model of computation, the resource bounds, and the kind of oracle access allowed.

The combination of the model of computation and resource-bounds is indicated by writing the correspondent complexity class as superscript to the $\leq$ sign. For example, $\leq^{\text{P}}$, and $\leq^{\text{L}}$ denote deterministic reducibilities running in polynomial time and logarithmic space respectively.

Usually the reduction only needs a restricted type of oracle access, namely many-one (m), bounded truth-table (btt), truth-table (tt) and Turing (T). The type of oracle access is denoted as a subscript to the $\leq$ sign: $\leq_{\text{m}}, \leq_{\text{btt}}, \leq_{\text{tt}}$, and $\leq_{\text{T}}$ respectively.

- A *many-one* (or Karp) reduction only makes one oracle query and outputs the answer to the oracle query. In case of languages $A$ and $B$, a many-one reduction

from $A$ to $B$ can be seen as a mapping $f : \Sigma^* \to \Sigma^*$ such that for any $x \in \Sigma^*$,

$$x \in A \text{ if and only if } f(x) \in B.$$

- A *bounded truth-table* reduction is a reduction which can ask only a bounded number of oracle queries. The answers can be combined in an arbitrary way to obtain the result of the reduction.

- A *truth-table* reduction is a reduction which can ask any number of queries but in a non-adaptative way, i.e., no query can depend upon the answers to previous questions.

- A *Turing* reduction is the most general type of reduction. There is no restriction on the oracle access whatsoever.

Most reductions used in computational complexity are transitive, they order problems with respect to their difficulty. So one is particularly interested in the maximal elements of this partial order.
We call a computational problem $A$ *hard* for a complexity class $\mathcal{C}$ under a given reducibility if every problem in $\mathcal{C}$ reduces to $A$. If in addition $A$ itself belongs to $\mathcal{C}$, we call $A$ complete for $\mathcal{C}$ or $\mathcal{C}$-complete. The type of reduction is often implicitly understood. In the context of the P versus NP question, $\leq^{\mathrm{P}}$-reductions will be used. The term NP-completeness refer to completeness for NP under $\leq^{\mathrm{P}}_{\mathrm{m}}$-reductions. If a polynomial time algorithm exists for any NP-complete problem, all problems in NP would be polynomial-time solvable. So the concept of NP-completeness gives us a method of locating problems in NP whose complexity is as difficult as any problem in NP.

**Definition 2.2.7 (SAT)** *A Boolean formula $f(x_1, ..., x_n)$ is* satisfiable *if there is a Boolean-valued truth assignment $a_1, ..., a_n$ such that $f(a_1, ..., a_n) = t$. Let SAT be the set of satisfiable Boolean formulas in conjunctive normal form (conjunction of disjunctions). The SAT* problem *is to decide whether or not a given Boolean formula is in SAT.*

In 1971, Cook [Coo71] proved the first natural problem, satisfiability, is NP-complete. Karp [Kar72] proved the NP-completeness of several famous combinatorial problems including traveling salesman and vertex cover. Since then many other problems has been proved to be NP-complete, see [GJ79].

### Oracles and the Polynomial Time Hierarchy

As we have already seen, sometimes we allow a Turing machine to use additional information essentially for "free". Depending on which information is actually provided, the Turing machine may be able to solve some problems more easily than before. Equivalently, we can also relativize complexity classes. As an example, the class $\mathrm{NP}^A$ consists of all languages recognizable by some polynomial time nondeterministic Turing machine with access to oracle $A$. We can also relativize complexity classes to other complexity classes. An example:

$$\mathrm{P}^{\mathrm{NP}} = \cup_{A \in \mathrm{NP}} \mathrm{P}^A.$$

Since SAT is NP-complete we have $\mathrm{NP} \subseteq \mathrm{P}^{\mathrm{NP}} = \mathrm{P}^{SAT}$. Now that we have defined an important deterministic complexity class, $\mathrm{P}^{\mathrm{NP}}$ one could ask for the correspondent nondeterministic complexity class, $\mathrm{NP}^{\mathrm{NP}}$. If we iterate this process we will get the polynomial-time hierarchy.

We recursively define the *hierarchy* as follows:

$$\begin{gathered}\Sigma_0 = \Pi_0 = \Delta_0 = \mathrm{P} \\ \Sigma_{i+1} = \mathrm{NP}^{\Sigma_i} \\ \Pi_{i+1} = \text{co-}\Sigma_{i+1} \\ \Delta_{i+1} = \mathrm{P}^{\Sigma_i}\end{gathered}$$

The *polynomial-time hierarchy* is the class $\mathrm{PH} = \cup_{i \geq 0} \Sigma_i$. We say the polynomial-time hierarchy collapses if $\mathrm{PH} = \Sigma_i$ for some $i$. If $\Sigma_i = \Sigma_{i+1}$ PH collapses also. It is generally believed that the hierarchy does not collapse. Some basic facts about the hierarchy:

$$\begin{gathered}\Delta_1 = \mathrm{P},\ \Sigma_1 = \mathrm{NP},\ \Pi_1 = \text{co-NP} \\ \mathrm{P} \subseteq \Sigma_1 \subseteq \Sigma_2 \subseteq \cdots \subseteq \mathrm{PH} \subseteq \mathrm{PSPACE} \\ \Sigma_i \subseteq \Delta_{i+1} \subseteq \Sigma_{i+1} \\ \Pi_i \subseteq \Delta_{i+1} \subseteq \Pi_{i+1}\end{gathered}$$

None of the inclusions is known to be proper.

Most theorems in complexity theory share the property that they remain true if, for any oracle $A$, we give to all Turing machines access to $A$. The reason why this happens is that almost all known *proof techniques* relativize, i.e., they remain valid when we give all programs access to the same oracle. This is the case for diagonalization arguments

and for simulations.
Baker *et al.* [BGS75] constructed an oracle $A$ (relativized world) such that $P^A = NP^A$ and another $B$ such that $P^B \neq NP^B$. These result is important because it indicate that we are unlikely to solve the $P \stackrel{?}{=} NP$ question by using the diagonalization method.
One can see the diagonalization method as a simulation of one Turing machine by another. The main idea is that the simulating machine can determine the behavior of the other machine and then behave differently. If we now give the same oracle to these machines, whenever the simulated machine queries the oracle, so can the simulator, and therefore the simulation can proceed as before. So as a consequence, any theorem proved about Turing machines by using only the diagonalization method would still hold is both machines were given the same oracle.
In particular, if we could prove that P and NP were different by diagonalization, we could conclude that they are different relative to any oracle. But Baker *et al.* [BGS75] have constructed an oracle such that P = NP, so the conclusion if false, and diagonalization is not sufficient to separate these two classes. Similarly, no proof relying on a simple simulation could show P = NP because that would show that they are the same relative to *any* oracle, but Baker *et al.* [BGS75] have constructed an oracle such that $P \neq NP$, so the conclusion if false. Therefore, settling this question requires non-relativizing proof techniques.

### Probabilistic Complexity Classes

Based on probabilistic Turing machines we will now study probabilistic complexity classes. We shall consider polynomial time bounds and different bounds on the error probability. We consider three types of probability error, *one-sided error, two-sided error and zero error probability.*

**Definition 2.2.8 (Random polynomial-time - RP)** *The complexity class RP is the class of all languages $L$ for which there exist a probabilistic polynomial-time Turing machine $M$ such that*

$$x \in L \text{ implies } Pr[M \text{ accepts } x] \geq \tfrac{1}{2}$$
$$x \notin L \text{ implies } Pr[M \text{ accepts } x] = 0$$

**Proposition 2.2.9** $NP \supseteq RP$

Due to Rabin-Miler and Solovay primality tests, "Given $n \in \mathbf{N}$, is $n$ composite" is an example of a problem in the class RP.

Note that there is a significant difference between nondeterministic Turing machines and probabilistic Turing machines. The first is not something that we could implement in practice (there may be only one good guess which make it accept), while the second can easily be implemented in practice by flipping coins.

One may argue that RP is too strict as it forces the machine to give correct answers for inputs that are not in the language.

**Definition 2.2.10 (Bounded probabilistic polynomial-time - BPP)** *The complexity class BPP is the class of all languages $L$ for which there exist a probabilistic polynomial-time Turing machine $M$ such that*

$$x \in L \text{ implies } Pr[M \text{ accepts } x] \geq \tfrac{2}{3}$$
$$x \notin L \text{ implies } Pr[M \text{ accepts } x] \leq \tfrac{1}{3}$$

We define this class with error probability $\frac{1}{3}$, but any constant between 0 and $\frac{1}{2}$ would yield an equivalent definition. This is a consequence of the following lemma:

**Lemma 2.2.11** *Let $\epsilon$ be any fixed constant between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$ a probabilistic polynomial-time Turing machine $M$ with error probability $\epsilon$ has an equivalent probabilistic polynomial-time Turing machine $N$ with an error probability $2^{-p(n)}$.*

It is obvious that one-sided error is a special case of two sided error, i.e., RP $\subseteq$ BPP. We do not know if BPP $\subseteq$ NP, however we know that it lies in the second level of the polynomial-time hierarchy.

**Theorem 2.2.12 ( [Lau83, Sip83])** BPP $\subseteq \Sigma_2^{\mathrm{P}} \cap \Pi_2^{\mathrm{P}}$.

**Definition 2.2.13 (Zero error probability - ZPP)** $L \in$ ZPP *if there exists a probabilistic polynomial-time Turing machine $M$, such that:*

$$\forall x, \quad Pr[M(x) = \perp] \leq \tfrac{1}{2}$$
$$\forall x, \quad Pr[M(x) = \chi_L(x)] + Pr[M(x) = \perp] = 1$$

*Where we denote the unknown answer sign as $\perp$.*

Some basic facts about probabilistic polynomial classes:

$$\mathrm{P} \subseteq \mathrm{ZPP} \subseteq \mathrm{RP} \subseteq \mathrm{BPP}$$

### Non-uniform Polynomial Time

The class of non-uniform polynomial time, P/poly , is the class of Turing machines that have access to *advice*, a small amount of additional information depending only on the input size. We will give two equivalent definitions of this class. The first based on a non-uniform family of circuits $\{C_n\}$, i.e., there is not necessarily any connection between a circuit of size $n$ and $n+1$.

**Definition 2.2.14** $L \in \mathrm{P/poly}$ *if there exists a sequence of circuits* $\{C_n\}$*, where for each* $n$*,* $C_n$ *has* $n$ *inputs and one output, and there exists a polynomial* $p(\cdot)$ *such that for all* $n$*,* $size(C_n) \leq p(n)$ *and* $C_n(x) = \chi_L(x)$ *for all* $x \in \Sigma^n$*.*

We can also define this class using polynomial time Turing machines with an *advice*. The non-uniformity is expressed in the fact that a different advice string may be defined for every different length of input.

**Definition 2.2.15** $L \in \mathrm{P/poly}$ *if there exists a polynomial-time two-input Turing machine* $M$*, a polynomial* $p(\cdot)$*, and a sequence* $\{a_n\}$ *of advice strings, where* $|a_n| \leq p(n)$*, such that for all* $x \in \Sigma^n$*,* $M(a_n, x) = \chi_L(x)$*.*

It can be proved that this definition is equivalent to the previous one. Note that we must restrict the length of the advice, as if we allow exponential long advice, $a_n$ could be a look-up table containing $\chi_L$, and so every language would be trivially in this class.
The class P can be viewed as the set of P/poly Turing machines with empty advice, so trivially $\mathrm{P} \subseteq \mathrm{P/poly}$. The second definition of P/poly is analogous to the class NP, more specifically the witness is analogous to the advice in P/poly. However this two classes differ in two ways:

1. For a fixed $n$, P/poly has a universal witness $a_n$ while for NP every $x \in \Sigma^n$ may have a different witness.

2. In the definition of NP, for every $x \notin L$ and for every witness $\omega$, $M(x, \omega) = 0$. However, this is not true for P/poly, i.e., we do not claim that there are not bad advice strings, we merely claim that there is a good advice string.

P/poly is not a realistic computational model. In fact it contains non-recursive languages, this is result of the non-uniformity inherent in P/poly. One do not even require the $\{a_n\}$ to be computable.

**Theorem 2.2.16** P/poly *contains non-recursive languages.*

**Proof.** (*Sketch*)

1. There exists unary languages which are non-recursive.
2. For every unary language $L$, $L \in$ P/poly.

◇

Another interesting property is that BPP languages have small size circuits.

**Theorem 2.2.17 ([Adl78])** BPP $\subseteq$ P/poly.

## 2.3 Kolmogorov Complexity

If someone claims that a finite sequence such as

10111001010001101001

or such as

01101001110010010111

was obtained by tossing a fair coin twenty times (0 and 1 correspond to the different sides of the coin) we would hardly be surprised. However, if we where told that the result of the flips was

00000000000000000000

then we would hardly trust the fairness of the experiment or even doubt about it. The question then is *why*?

We definitely have some intuitive notion of a random sequence. The first two sequences are *perceived* to be random while the third is regarded as nonrandom. So what do the words perceived to be random mean? Classical probability theory does not give an answer to this question. One could argue that the probability of the third sequence, $2^{-20}$, is too small. But the fact is that the first two sequences have exactly the same probability.

Kolmogorov complexity provide us with a sound basis for our intuition of randomness. Kolmogorov [Kol65], Solomonoff [Sol64] and Chaitin [Cha66] independently defined the complexity of an individual object (string) $x$ as the length of the shortest program that produces $x$.

An effective enumeration of Turing machines $T_1, T_2, \cdots$, induces an effective enumeration of partial recursive functions $\phi_1, \phi_2, \cdots$ such that $T_i$ computes $\phi_i$ for all $i$. In the literature Kolmogorov complexity is defined in terms of *partial recursive functions*, or in terms of *Turing machines*. These two versions of Kolmogorov complexity are in fact the *same* in the absence of time bounds.

**Definition 2.3.1 (Conditional Kolmogorov complexity)** *Let $\varphi$ be a partial recursive function on binary strings. For any pair of strings $x, y \in \{0,1\}^*$, the complexity of $x$ relative to $y$ with respect to $\varphi$ is defined as*

$$C_\varphi(x|y) = \min\{|p| : \varphi(p, y) = x\}.$$

The complexity measure $C_\varphi$ depends on $\varphi$. However, we can achieve a sort of independence using a universal partial recursive function. The following invariance theorem is the cornerstone for the subsequent development of the theory, showing that Kolmogorov complexity is an intrinsic property of strings. Without it the concept of Kolmogorov complexity would be virtually useless.

**Theorem 2.3.2 (Invariance theorem)** *There is a universal partial recursive function $\Phi$ such that for any partial recursive function $\varphi$, there is a constant $c_\varphi$, for which*

$$C_\Phi(x|y) \leq C_\varphi(x|y) + c_\varphi, \text{ for all } x, y.$$

This does not necessarily means that the universal description gives the shortest description in each case, but that no other description method can improve it infinitely often.

Fix a universal $\Phi$ and write $C(x|y)$ instead of $C_\Phi(x|y)$. This particular $\Phi$ is called the *reference function* for $C$. We also fix a particular Turing machine $U$ that computes $\Phi$ and call $U$ the *reference machine.* When $y = \epsilon$, the empty string, we write $C(x)$, for the *unconditional Kolmogorov complexity.* Let $x \in \{0,1\}^n$, then the complexity measure $C(x)$ gives the minimum length of a program, $p$, for $x$ that contains information about distribution of characters in $x$ as well as the length of $x$. When the length of $x$ is known we use $C(x|n)$, which reflects only information about the distribution of characters in $x$. This distinction is dramatic at the low complexity end of the scale where the information needed to determine the distribution is less than $\log n$. Comparison between $x \in \Sigma^n$ and $y \in \Sigma^n$ regarding the distribution of symbols, may be lost in the need to report that each is of length $n$.

Some basic properties of Kolmogorov complexity:

- $C(x) \leq |x| + c$ for all $x$.
- For any partial computable $\phi$, $C(\phi(x)) \leq C(x) + c_\phi$, where $c_\phi$ is the length of a description of $\phi$.
- $C(x|y) \leq C(x) + c$.

Now let us recall the question stated in the beginning of the section: "So what do the words, perceived to be random, mean?" The idea is that a string is random if it has no short description. Using $C(x)$ we can formalize this idea in the following way.

**Proposition 2.3.3** *For all $n \in \mathbb{N}$, there is at least one string $x \in \Sigma^n$ with $C(x) \geq n$.*

**Definition 2.3.4** *For each constant c, a string $x \in \{0,1\}^n$ is said to be c–incompressible (or algorithmically random) if $C(x) \geq n - c$.*

**Theorem 2.3.5** *For all $k$ and $n$, $|\{x \in \Sigma^n : C(x) \geq n - k\}| \geq 2^n(1 - 2^{-k}) + 1$.*

**Proof.** The number of programs of size smaller than $n-k$ is $2^{n-k}-1$. So the number of programs of length greater than $n - k$ is $2^n - 2^{n-k} + 1 = 2^n(1 - 2^{-k}) + 1$. $\diamond$ The are countless examples of proofs which use the existence of incompressible strings at the center of their arguments. Many can be found in [LV97b].

**Example 2.3.6**

***There are infinitely many primes.***

*Suppose not. Then assume $p_1, \cdots, p_k$ is the list of all primes, for some $k \in \mathbf{N}$. Let $m = p_1^{e_1} \cdots p_k^{e_k}$ be algorithmically random with length $n$. We can describe $m$ by $\langle e_1, \cdots, e_k \rangle$. We claim that this gives a short description of $m$. Each of the exponents is smaller than the logarithm of $m$, therefore they can be described using $\log \log m$ symbols. Hence*

$$|\langle e_1, \cdots, e_k \rangle| \leq 2k \log \log m.$$

*As $m \leq 2^{n+1}$ we have,*

$$|\langle e_1, \cdots, e_k \rangle| \leq 2k \log(n+1),$$

*so*

$$C(m) \leq 2k \log(n+1) + c.$$

*For large enough $n$, this contradicts $C(m) \geq n$, which follows from the fact that $m$ is random.*

One could claim that the proof above is more complex than the original one. However, the following result is definitely easier than the original one.

**Example 2.3.7**

*Let $p_m$ be the $m$th prime.* ***How big is*** *$p_m$?*

*Let $p_i$ be the largest prime that divides $m$. Then we can describe $m$ by specifying $p_i$ and $\frac{m}{p_i}$ because we can compute $p_i$ given $i$. For a random $m$ we have, (from Theorem 2.3.5 most strings are random)*

$$\begin{aligned} C(m) &\leq C(\langle i, \tfrac{m}{p_i} \rangle) \\ &\leq 2 \log |i| + |i| + |\tfrac{m}{p_i}| \end{aligned}$$

*so*

$$\begin{aligned} \log m &\leq 2 \log \log i + \log i + \log m - \log p_i \\ \log p_i &\leq 2 \log \log i + \log i \\ p_i &\leq i (\log i)^2 \end{aligned}$$

*The classical result is that the $i$th prime is below $i \log i$, so the above is very close.*

Can we compute $C(x)$? One could try to compute $C(x)$ by running all programs $p$ with $|p| < |x| + O(1)$ and find the shortest one generating $x$. However, this does not work

because some programs do not halt. One of the consequences of the undecidability of the halting problem is the non-computability of Kolmogorov complexity. In fact, no algorithm can compute $K$ or even any its lower bounds except $O(1)$.

The non-computability of Kolmogorov complexity is an example of the Berry paradox:

*The smallest number that cannot be* **defined** *in fewer than twenty words.*

No number can be a solution since the defining expression itself is less than twenty words long. The problem has to do with the notion *defined*, which is too powerful to be used without a strict meaning. Formalizing the notion of *definition* as the shortest program from which a number can be computed by the reference machine $U$, it turns out that the quoted statement is not an *effective description.* E. F. Beckenbach pointed out a similar problem in the classification of numbers as dull or interesting; the smallest dull number must be interesting.

A similar argument proves that $C$ is not computable. Suppose that some algorithm $A$ computes a lower bound for $C(x)$. We can use $A$ to compute $\phi(n)$, a function that finds $x$ with $n < A(x) < C(x)$. But as $C(x) < C_\phi(x) + O(1)$ and $C_\phi(\phi(n)) \leq |n|$, we have

$$n < C(\phi(n)) < |n| + O(1) = \log O(n) \ll n$$

a contradiction. So, $C$ and its non-constant lower bounds are not computable.

**Theorem 2.3.8** *The function $C(x)$ is not partial recursive.*

As we have see $C(x)$ is not computable, however we can approximate it.

**Theorem 2.3.9** *There is a total recursive function $\phi(t, x)$, monotonic decreasing in $t$, such that $\lim_{t\to\infty} \phi(t, x) = C(x)$.*

Usually the busy beaver function, $BB(n)$, is defined to be the largest number which can be computed by an $n$-state Turing machine. Although many variations on the definition of busy beaver have been used, we recast the original definition (see Daley [Dal82]) as follows:

**Definition 2.3.10** *The* busy beaver *function is defined $BB : \mathbf{N} \to \mathbf{N}$ as*

$$BB(n) = \max_{p:|p|\leq n} \{\textit{Running time of } U(p) \textit{ when defined}\}$$

Note that, $BB$ grows faster than any recursive function, so its noncomputable.

### Symmetry of information

Kolmogorov and Levin (see [ZL70b]) independently proved one of the most powerful results in Kolmogorov complexity: *symmetry of information.* One can use Kolmogorov complexity to measure the amount of information of one string $x$ contained in another string $y$. Symmetry of information states that within a logarithmic additive factor the amount of information of $x$ contained in $y$ equals the amount of information of $y$ contained in $x$.

**Definition 2.3.11** *The* algorithmic information *about $y$ contained in $x$ is*

$$I_C(x : y) = C(y) - C(y|x)$$

Since $C(y) \geq C(y|x) + O(1)$ we have $I_C(x : y) \geq 0$.

Let $C(x, y) = C(\langle x, y \rangle)$. We can describe $\langle x, y \rangle$ by giving a shortest description of $y$, a shortest description of $x$ given $y$ and an indication of where to separate the two items. Let $n = \max\{|x|, |y|\}$, then we have

$$C(x, y) \leq C(y) + C(x|y) + O(\log n)$$

In order to prove that this inequality is tight, let us look at a simple combinatorial argument. Let $A$ be a finite set of pairs of strings. For any string $y$ we consider the set $A_y$ defined as

$$A_y = \{x : \langle x, y \rangle \in A\}$$

The number of elements in $A_y$, depends on $y$, and is 0 for all $y$ outside some finite set. Clearly,

$$\sum_y |A_y| = |A|.$$

Therefore, the number of $y$ such that $|A_y|$ is big, is limited:

$$|\{y : |A_y| \geq c\}| \leq \frac{|A|}{c}$$

for any $c$.

**Theorem 2.3.12 (Symmetry of information)** *Let* $n = \max\{|x|, |y|\}$. *We have:*

$$C(x|y) + C(y) \leq C(x, y) + O(\log n)$$

**Proof.** Let $C(x, y) = a$. Consider the set $A$ of all pairs $\langle x, y\rangle$ that have complexity at most $a$. Let $b = \lfloor \log |A_y| \rfloor$. To describe $x$ when $y$ is known we need to specify $a, b$ and the index of $x$ in $A_y$. Note that $A_y$ can be enumerated effectively if $a$ and $b$ are known, since $C$ is enumerable from above from Theorem 2.3.9. This index as $b + O(1)$ bits, therefore

$$C(x|y) \leq b + O(\log a + \log b).$$

On the other hand, the set of all $y'$ such that $|A_{y'}| \geq 2^b$ consists of at most $\frac{|A|}{2^b} = O(2^{a-b})$ elements, and can be enumerated when $a$ and $b$ are known. The given $y$ belongs to this set, therefore, $y$ can be described by $a, b$ and the index of $y$ in the set, and $C(y) \leq a - b + O(\log a + \log b)$. Therefore,

$$C(y) + C(x|y) \leq a + O(\log a + \log b).$$

◇

**Corollary 2.3.13** $I(x : y) = I(y : x) \pm O(\log n)$, *where* $n = \max\{|x|, |y|\}$.

### Prefix-free Kolmogorov complexity

The Kolmogorov complexity as we have defined has some drawbacks: *(i)* it is not sub-additive ($C(x, y) \leq C(x) + C(y)$ holds only to within a logarithmic term in $C(x)$ or $C(y)$), *(ii)*it is not monotonic over prefixes, *(iii)* it can not be used, as Solomonoff [Sol64] originally conjectured, to assign a universal *prior probability* to each finite binary string (manly because for a given $x$, the series $\sum_{p:U(p)=x} 2^{|p|}$ diverges).

The divergence of this series almost destroyed Solomonoff work. However Levin [Lev74], by considering the complexity induced by Turing machines with a set of programs in which no program is a proper prefix of another program, was able to solve this problem and the sub-additive one.

**Theorem 2.3.14 (Kraft's Inequality)** *If* $A$ *is a prefix-free set then*

$$\sum_{x \in A} 2^{-|x|} \leq 1.$$

**Model:** A *prefix-free* machine is a Turing machine with a one-way input tape (the input head can only read from left to right), a one-way output tape and a two-way work tape.

**Definition 2.3.15** *A* partial recursive prefix function $\phi : \Sigma^* \to \mathbf{N}$ *is a partial recursive function such that if* $\phi(p) \downarrow$ *and* $\phi(q) \downarrow$, *then* $p$ *is not a proper prefix of* $q$.

There are effective enumerations of prefix machines $T_1, T_2, \cdots$, computing exactly the partial recursive prefix functions $\phi_1, \phi_2, \cdots$.

**Theorem 2.3.16** *Every partial recursive prefix function can be simulated by a prefix machine, and there is a universal prefix machine.*

**Proof.** (*Sketch*) The proof of this theorem is based on a construction given in [Cha75] by Chaitin. Let $\phi$ be a partial recursive function, and the domain of $\phi$ prefix-free. The corresponding prefix machine acts as follows. Read a bit of the input $z$. Before reading any more, simulate $\phi$ simultaneously on all $y$ such that $z$ is a prefix of $y$, until $\phi(y)$ halts, if ever. If $y = z$, output $f(y)$, otherwise read the next input bit.
This argument produces a prefix machine for each partial recursive prefix function, and a universal machine. The universal machine on input $0^{|p|}px$ uses $p$ as the program for some $\phi$ and simulates the above prefix machine for $\phi$ on input $x$. ◇

**Theorem 2.3.17** *There is a partial recursive prefix function* $\Phi$ *such that for any partial recursive prefix function* $\phi$ *there is a constant* $c_\phi$ *such that* $C_\Phi(x|y) \leq C_\phi(x|y) + c_\Phi$.

**Definition 2.3.18** *We fix an additively optimal partial recursive prefix function* $\Phi$ *as the standard reference, and a prefix machine* $U$ *as the reference prefix machine, and define the* prefix complexity *of* $x$, *conditional to* $y$, *by* $K(x|y) = C_\Phi(x|y)$ *for all* $x$.

Note that now we can assign a measure $\mu(x) = 2^{-K(x)}$, since by Kraft's inequality $\sum_x \mu(x) < 1$. This measure assigns a weight to each string according to its Kolmogorov complexity. The shorter is the description, the heavier is the weighting.

As we need a prefix-free way of describing $x$ we have

$$K(x) \leq 2 \log |x| + |x| + c.$$

We can improve this bound by using more efficient prefix codes to

$$K(x) \leq 2 \log \log |x| + 2 \log |x| + |x| + c.$$

and so on.

**Theorem 2.3.19**

*(i) For each $n$, $\max\{K(x) : |x| = n\} = n + K(n) + O(1)$.*

*(ii) For each fixed constant $r$, the number of $x \in \Sigma^n$ with $K(x) \leq n + K(n) - r$ does not exceed $2^{n-r+O(1)}$.*

So we can conclude that the great majority of all $x \in \Sigma^n$ has complexity larger than $n$. Notice that by using the prefix complexity in the symmetry of information theorem we get a sharper bound.

**Theorem 2.3.20** $K(x, y) \leq K(x) + K(y) + c$.

**Resource-bounded Kolmogorov complexity**

As we have seen Kolmogorov complexity measures the amount of randomness or information contained in a string. Unfortunately, this definition of randomness puts no limits on the time of computation to obtain the given string from its compressed encoding, and therefore makes randomness an undecidable property.

One must be careful defining time bounded prefix Kolmogorov complexity. As we have seen, we can define prefix Kolmogorov complexity based on prefix machines and on partial recursive prefix functions. These notions are equivalent because, as Chaitin [Cha75] showed, every partial recursive function can be simulated by a prefix machine. However, this simulation is not efficient, and so the time bounded equivalence between these notions is an open problem in the area.

Adding time bounds to prefix Kolmogorov complexity based on prefix machines changes very little. There exist efficient universal prefix machines, so proving and invariance theorem for time-bounded Kolmogorov complexity based on prefix machines is straightforward. Adding time bounds to prefix Kolmogorov complexity based on

partial recursive prefix functions is more problematic. In fact, it is not known whether this version of time-bounded Kolmogorov complexity satisfies an invariance theorem. We could use Chaitin's construction [Cha75] (used in the proof of Theorem 2.3.16), however it dramatically increase the running time of the original machine. So, answer this question may be difficult. In fact, Juedes and Lutz [JL00], have studied this question and proved the following result:

**Theorem 2.3.21** *Every partial recursive prefix function can be simulated by a prefix machine if and only if* P = NP.

**Definition 2.3.22 (Time-bounded Kolmogorov complexity)** *Let $M$ be a prefix machine, and $t$ a time constructible function. For any pair of strings $x, y \in \{0,1\}^*$, the time-bounded complexity of $x$ relative to $y$ with respect to $M$ is defined as*

$$K_M^t(x|y) = \min\{|p| : M(p, y) = x \text{ in } t(|x| + |y|) \text{ steps}\}.$$

Clearly, for total recursive functions $t(n)$ the function $K_M^t$ is total recursive as well. We could also define a space-bounded version, but in this work we will not use it. The following results can be found in [LV97b].

**Theorem 2.3.23** *There exists an* efficient universal *prefix machine.*

The invariance theorem for the time-bounded Kolmogorov complexity, is considerably weaker.

**Theorem 2.3.24** *Let $x \in \Sigma^*$ and $t$ be a time constructible function. There exists a prefix machine $U$ such that for all prefix machine $M$ there exists a constant $c$ such that*

$$K_U^{c \cdot t \cdot \log t}(x) \leq K_M^t(x) + c.$$

As usual, we now fix a universal prefix machine $U$ and let $K^t = K_U^t$.

As Kolmogorov stated (see [LM93]), the problem of whether symmetry of information holds in time bounded environments has interesting connections to complexity theory. In fact, we would like to have efficient access to information in a string. Longpré

and Mocas [LM93] and Longpré and Watanabe [LW95] have examined symmetry of information for time-bounded Kolmogorov complexity. They showed that if certain kinds of one-way functions exist then symmetry of information fails in the polynomial-time cases. Intuitively, a one-way function is a function that is easy to compute, but such that its inverse is hard to compute.

### Distinguishing complexity

In a resource bounded setting, instead of defining the shortest program that *produces* a string $x$, we could also consider the shortest program that *distinguishes* the string $x$ from all other strings. In the unbounded case the two measures coincide, as we could run through all possible strings until we find one accepted by the program, and print it. For a detailed analysis of this measure see [BFL01].

This measure of complexity was introduced by Sipser [Sip83], who used it to show that $BPP$ is in the polynomial hierarchy.

**Definition 2.3.25 ([Sip83])** *Let $U$ be some fixed universal Turing machine. For any pair $x, y \in \{0,1\}^*$, the t-time-bounded distinguishing complexity of $x$ relative to $y$ is:*

$$CD^t(x|y) = \min_p\{|p| : U(p,x,y) = 1 \text{ and } U(p,z,y) = 0 \text{ for all } z \neq x \text{ and for all } z\ U(p,z,y) \text{ runs in at most } t(|z|+|y|) \text{ steps }\}.$$

The exact difference between $C^t(x|y)$ and $CD^t(x|y)$ is not known. It is conceivable that both measures are always very close. We can easily prove that, for all polynomial $p$ there is a polynomial $q$ such that

$$CD^q(x|y) \leq C^p(x|y) + c \tag{2.1}$$

as a program generating $x$ trivially distinguish $x$. The converse, for all polynomial $p$ there is a polynomial $q$ such that

$$C^q(x|y) \leq CD^p(x|y) + c \tag{2.2}$$

is much harder. In fact it is even unlikely, since Fortnow and Kummer [FK96] showed that if (2.2) holds then the promise problem $(1SAT, SAT)$ can be solved in polynomial time.

Recall that $(1SAT, SAT) \in P$ if there is a deterministic polynomial-time algorithm which accepts all Boolean formulas with a unique satisfying assignment, and rejects all Boolean formulas which are not satisfiable. $(1SAT, SAT) \in P$ implies $NP = RP$, so in particular factoring is in $P$.

**Theorem 2.3.26 (Fortnow-Kummer)** *$(1SAT, SAT) \in P$ if and only if for every polynomial $p_1$ there is a polynomial $p_2$ and a constant $c$ such that for any string $x$ of length $n$, and any string $y$*

$$C^{p_2}(x|y) \leq CD^{p_1}(x|y) + c \log n.$$

Sipser used distinguishing complexity to answer the question of how much information is needed to distinguish a given string from all other strings in a given set.

Kolmogorov complexity give the following answer to this question.

**Lemma 2.3.27** *Let $A$ be a computable set. For all $n$ and for all $x \in A \cap \{0,1\}^n$,*

$$C(x) \leq \log |A \cap \{0,1\}^n| + O(\log n)$$

Sipser proved the following theorem with the aid of a polynomial-long random string.

**Theorem 2.3.28 ([Sip83])** *For every set $A \in P$, there is a polynomial $p$ and a constant $c$ such that for every $n$, for most $r$ of length $p(n)$, and for every $x \in A \cap \{0,1\}^n$,*

$$CD^p(x|r) \leq \log |A \cap \{0,1\}^n| + c \log n.$$

Buhrman and Fortnow showed how to eliminate $r$ at the cost of doubling the complexity.

**Theorem 2.3.29 ([BF97])** *For every set $A \in P$, there is a polynomial $p$ and a constant $c$ such that for all strings $x \in A \cap \{0,1\}^n$,*

$$CD^p(x) \leq 2 \log |A \cap \{0,1\}^n| + c \log n.$$

The technique used in the proof of the previous theorem consider values modulo a prime, it resembles hashing via the division method. It is based used on the following lemma.

**Lemma 2.3.30** *Let $S = \{x_1, \cdots, x_d\} \subseteq \{1, \cdots, 2^n - 1\}$. For all $x_i \in S$ and at least half of the primes $p \leq 4dn^2$, $x_i \not\equiv x_j \mod p$ for all $i \neq j$.*

**Proof.** *(Sketch)* For each $x_i, x_j \in S$, $i \neq j$, it holds that for at most $n$ different prime numbers $p$, $x_i \equiv x_j \mod p$ by the Chinese Remainder Theorem. For $x_i$ there are at most $dn$ primes $p$ such that $x_i \equiv x_j \mod p$ for some $x_j \in S$. The Prime Number Theorem states that for any $m$ there are approximately $\frac{m}{\ln m} > \frac{m}{\log m}$ primes less than $m$. There are at least

$$\frac{4dn^2}{\log(4dn^2)}$$

primes less than $4dn^2$. So at least half of these primes $p$ must have $x_i \not\equiv x_j \mod p$ for all $i \neq j$. ◇

Recently Buhrman, Laplante and Miltersen [BLM00] have proved that the constant factor 2 in Theorem 2.3.29 is optimal in relativized worlds.

**Theorem 2.3.31 ([BLM00])** *For any polynomial time bound $t$ there is an input length $n$ for which the following holds. Any random set of size $r$ contains a set $A \subseteq \{0,1\}^n$ of size $r^{\frac{1}{3}}$ for which at least some $x \in A$ has $CD^{t,A}(x) \geq 2\log|A| + \log\log r - \log 100 - \log\log|A|$.*

Fortnow and Laplante showed that the factor of 2 can be removed for all but a small fraction of the strings in $x \in A \cap \{0,1\}^n$. The closer we get to all strings, the worse the approximation is.

**Theorem 2.3.32 ([FL98])** *For all but an $\epsilon$ fraction of the $x$ in $A \cap \{0,1\}^n$, $CD^{poly,A}(x) \leq \log|A| + (\log\frac{n}{\epsilon})^{O(1)}$*

Fortnow [For01] made the following interesting observation:

**Theorem 2.3.33** *The statement, for all polynomial $p$ there is a polynomial $q$ such that*

$$C^q(x|y) \leq CD^p(x|y) + c\log|x|$$

*is equivalent to*
*"There is a polynomial-time computable function $f$ such that for all formulas $\varphi$ with exactly one satisfying assignment, $f(\varphi)$ outputs the assignment."*

The existence of such a function is thought to be very unlikely; indeed, it is thought to be only slightly weaker than P = NP.

### Generalized Kolmogorov complexity

Hartmanis [Har83] introduced a generalization of Kolmogorov complexity where the computation time and space are taken into account. The advantage of this approach is that it not only classifies strings as random or not, but measures the amount of randomness detectable in a given time. This allow us to study how computations change the amount of randomness of finite strings and thus establishes a direct link between computational complexity and Kolmogorov complexity.

**Definition 2.3.34 (Hartmanis)** *Let $f$ and $g$ be time constructible functions.*

$$K[f(n), g(n)] = \{x : \text{ exists } y \text{ such that } |y| \leq f(|x|) \text{ and } U(y) = x \text{ in time } \leq g(|x|)\}$$

For instance, $x \in \Sigma^n$ is in the Kolmogorov class $K[\log n, n^3]$ if there exists a string $y$ of length smaller or equal than $\log n$ from which $x$ can be computed in $n^3$ steps.

The advantage of this approach is that it measures the amount of randomness "detectable" in a given time. It measures how far $(f(n))$ and how fast $(g(n))$ a string can be compressed. However running time and program length are not combined in any way.

Now consider the following problem: given a satisfiable Boolean formula $\phi$, find a truth assignment that satisfies it. When searching trough all the $2^n$ possible assignments, what is the optimal search strategy?

To answer this question Levin [Lev73b] introduced a useful variant of Kolmogorov complexity weighing program size and running time.

**Definition 2.3.35 ([Lev73b])** *Let $M$ be a Turing machine. For any pair of strings $x, y \in \{0,1\}^*$, the complexity of $x$ relative to $y$ with respect to $M$ is defined as*

$$Kt_M(x|y) = \min\{|p| + \log t : M(p, y) = x \text{ in at most } t \text{ steps}\}.$$

The answer to the previous question is to consider each string $x \in \{0,1\}^n$ in order of increasing $Kt(x|\phi)$. This measure discards some intractable descriptions, and as a result becomes computable. $Kt$ complexity is closely related to generalized Kolmogorov complexity; for $x \in \{0,1\}^*$ if $x \in K[m - \log t, t]$, then $Kt(x) \leq m$.

**Proposition 2.3.36** $Kt(x) \leq f(|x|) \Rightarrow x \in K[f(|x|), 2^{f(|x|)}] \Rightarrow Kt(x) \leq 2f(|x|)$.

We finish this section, and motivate the following chapter, with a citation from Li and Vitányi book:

*At present, prefix-code based complexity is often considered as some sort of standard algorithmic complexity. Lest the reader be deluded into the fallacy that this is the most perfect of all possible worlds, we state that different applications turn out to require different versions of complexity, and all of these are natural for their own purposes.*

*(Li and Vitányi [LV97b])*

# 3

# Computational Depth

Kolmogorov complexity measures the amount of information contained in a string $x$ as the length of a shortest description of $x$. Random strings are incompressible, as the shortest description of this strings is the string itself. They are therefore deemed to contain a lot of information. However, random information may not be very useful from a computational point of view.

Intuitively, computational depth measures the amount of nonrandom or useful information in a string. Formalizing this intuitive notion is tricky. A computationally deep string $x$ should take a lot of effort to be constructed from its shortest description. Incompressible strings are trivially constructible from their shortest description, and therefore computationally shallow.

After some attempts Bennett [Ben88] formally defined the $s$-significant logical depth of an object $x$ as the time required by a standard universal Turing machine to generate $x$ by a program $p$ that cannot itself be obtained from a program that is $s$ or more bits shorter than $p$. We consider logical depth as one instantiation of this general theme, and propose several other variants. Which one is most appropriate depends on the problems we want to apply it to.

We start this chapter describing the previous work, potential [Ade79] and logical depth [Ben85, Ben86, Ben88] then we will introduce three other measures capturing the intuition behind computational depth [AFvM01].

The first notion of computational depth we propose is the difference between a time-

bounded Kolmogorov complexity and traditional Kolmogorov complexity. From this measure we define *shallow sets* and show that, in a computational setting, shallow sets provide very little useful information: Every computable set that reduces to a shallow set also reduces to a sparse set.

Our second variant is the difference between two time bounded complexity measures namely, $C^{t_1}(x)$ and $CD^{t_2}(x)$ where $t_1, t_2$ are polynomials. We establish an analog to Bennett's slow growth law for this measure. We also show that if a nonnegligible fraction of the satisfying assignments of a Boolean formula $\phi$ have small depth then we can find a satisfying assignment of $\phi$ in quasi-polynomial probabilistic time.

Finally, we present a third variant based on Levin's $Ct$ complexity. We argue that it forms a closely related but more manageable version of Bennett's logical depth.

## 3.1 Previous Work

### 3.1.1 Potential

This subsection is based on the work of Adleman [Ade79] and Section 7.4.2 of Li and Vitányi book [LV93]. Consider the following equation in chemistry:

$$2H_2 + O_2 \rightleftharpoons 2H_2O$$

does it share any similarities with the following equation in number theory

$$p \times q = pq?$$

Converting $H_2$ and $O_2$ to $H_2O$ (oxidation) is quite rapid; the converse (electrolysis) is quite slow. Adleman has noted that we could see the oxidation process as prime multiplication, the electrolysis as prime factorization and the previous interpretation would also hold, by know methods, multiplication is fast and factorization slow. After note this connection Adleman naturally asked if the chemists also have an NP = P analog question:

> *"Are there fast methods for reduction than electrolysis, or are there reasons in principle why all methods must be slow?"*

Fortunately for the chemist there are reasons in principle, potential energy. Converting $H_2$ and $O_2$ to $H_2O$ generates a great amount of energy; the converse process requires a large amount of energy to be pumped back in. So one could ask if the analogy still holds, i.e., is there a notion of storing potential in number? Adleman has answer this question identifying time consuming with energy. Computing a large composite number from two primes costs a small amount of time however the converse is likely to be difficult and time consuming.

**Definition 3.1.1** *Let $x, y \in \{0,1\}^*$ and $|x| = n$. A string $x$ is $k$-potent relative to $y$ if $k$ is the least positive integer such that $Kt(x|y) \leq k \log n$. When $y = \epsilon$ we say that $x$ is $k$-potent.*

The following Lemma relates the potential with the Hartmanis's notion of generalized Kolmogorov complexity.

**Lemma 3.1.2** *Let $x, y \in \{0,1\}^*$ and $|x| = n$. A string $x$ is $k$-potent relative to $y$ if and only if $k'$ is the least integer such that*

$$x \in K[k' \log n, n^{k'}|y]$$

**Example 3.1.3**

- *For almost all $n$, $1^n$ is 2-potent.*
- *For all $k$, almost all incompressible $x$ are not $k$-potent.*

Although no (honest) function can map an arbitrary long nonrandom string into a random one, computable functions which infinitely often produce outputs of higher potential do exist. Such functions are called *inflating.*

**Definition 3.1.4** *A function from $\{0,1\}^*$ to $\{0,1\}^*$ is inflating if and only if for all integers $k$ there are infinitely many $x$ such that $f(x)$ is not $k$-potent relative to $x$.*

The following theorem takes advantage of the log function in Levin's $Kt$ complexity.

**Theorem 3.1.5** *Let $h$ be a honest function, such that given $x$ and $y$ we can decide in polynomial time whether or not $y = h(x)$. Then $h \in \mathrm{P}$ if and only if $h$ is not inflating.*

**Proof.** Let $n = |h(x)|$.

$[h \in P \Rightarrow h$ is not inflating.]

By hypothesis $h \in P$, i.e., there is a constant size algorithm computing $h(x)$ in time $t(m)$ polynomial in $m = l(x)$. So

$$Kt(h(x)|x) \leq c + 2\log t(m).$$

Since $h$ is honest, we have $m \leq n^k$ for some fixed $k$ independent of $x$. Thus

$$Kt(h(x)|x) \leq c + 2k\log t(n).$$

[$h$ is not inflating $\Rightarrow h \in P$]

Assume $h$ is not inflating, then there is a $k$ and for all $x$, $Kt(h(x)|x) \leq k\log m$. Now run all programs $p$ such that $Kt(p|x) \leq k\log m$ dovetail fashion and check for each output whether $y = h(x)$. There are $m^{2k}$ such programs, each of which runs in polynomial time, and by assumption the verification process also takes polynomial time. So we can find $h(x)$ in polynomial time and $h \in P$. ◇

As a consequence we get that, factoring is difficult if and only if multiplication infinitely often takes highly potent numbers and produce relatively low potent products, which are hard to factor because because the potential must be pumped back. In fact searching for factors of a number by enumerating candidates in order of potency makes sense. The last theorem shows that the difficulty of factoring is not a global phenomenon but rather is a local one which we might hope to see in particular inputs and outputs.

### 3.1.2 Logical Depth

This subsection is based on Bennett's work [Ben85, Ben86, Ben88] and Section 7.7 of [LV97b]. Inspired by the work of Hartmanis [Har83] we obtain some results using the set of all small depth strings of length $n$. Faced with a fairly detailed description of the positions of the moon and planets every day for the last 10 years, we might attribute to it a high effective complexity until we learn that it can be generated from a very simple equation of motion and initial conditions from which it was calculated. Although highly compressible, as is the program generating 01010101..., we would like to say that the information embedded in the description of the position of the moon and planets is richer than the one embedded in 01010101.... This observation is essentially

important when one considers sequences generated by recursive programs. Indeed, the information on a given digit could be extremely tedious to obtain, and could be obtained after a very long computation time. In this respect the knowledge of the precise value of this digit, the "information" on it would seem to be very high, had we another quick means to obtain it. Bennett has formalized this aspect of sequences, in terms of "logical depth".

After some attempts to define this new complexity measure, Bennett [Ben88] formally defined the $s$-significant logical depth of an object $x$ as the *time* required by a standard universal Turing machine to generate $x$ by a program $p$ that cannot itself be obtained from a program that is $s$ or more bits shorter than $p$.

We will start by enumerating all the attempts and comment the advantages and disadvantages of each one of them. As we will see in this Chapter, it is quite subtle to give a formal definition of *computational depth* that satisfies the intuitive notion. Based on the intuition we could argue that to define computational depth it is enough to consider the running time of the minimal program, this was Bennett's first attempt.

**Definition 3.1.6 (Attempt 1)** *A string's depth might be defined as the execution time of its minimal program.*

The main problem behind this definition is *stability.* There are strings for which the minimal program is just a few bits smaller than some much faster program. So we are lead to think that we should not only consider the *running time* but also the *program size* in the definition of computational depth. This lead us to Bennett's Attempt 2. A string depth at a significance level $s$ is defined as the amount of time complexity allowing $s$ bits of buried redundancy.

**Definition 3.1.7 (Attempt 2)** *A string's depth at a significance level $s$ is defined as the time required to compute the string by a program no more than $s$ bits longer than the minimal program.*

In Attempt 2, Bennett has relaxed the requirement "minimal program" for "almost minimal program" which turns the definition stable. But this definition was not yet satisfactory because of the way it treats multiple programs of the same length. If $2^k$

distinct $(n+k)$-bit programs compute the same output then they should be given the same weight as one $n$-bit program; however this is not the case in the present definition, they are given as much weight as one $n+k$-bit program. This lead us to Bennett's Attempt 3 taking explicitly the algorithmic probability into account, weighing all possible causes of emergence of $x$ appropriately.

**Definition 3.1.8 (Attempt 3)** *The* depth *of a string $x$ at a significance level $\epsilon = 2^{-b}$ is*

$$depth_\epsilon(x) = \min\{t : \frac{Q^t_U(x)}{Q_U(x)} \geq \epsilon\}$$

*where $Q_U(x) = \sum_{U(p)=x} 2^{-|p|}$, $Q^t_U(x) = \sum_{U^t(p)=x} 2^{-|p|}$ and $U^t(p) = x$ means that $U$ computes $x$ within $t$ steps and halts. A string $x$ is $(b,d)$-deep if $d = depth_\epsilon(x)$ and $\epsilon = 2^{-b}$.*

One natural question is whether there exists a string such that it is deep under Attempt 2, deterministic deep, but shallow under Attempt 3, probabilistic shallow. This questions do not relativize uniformly. Relative to a random oracle this measures are identical however relative to a deliberately designed oracle (hiding information from deterministic computations) they can be very different.

Although Attempt 3 capture Bennett's intuition of Logical Depth he introduced another definition for technical reasons.

**Definition 3.1.9** *The depth of a string $x$ at a significance level $s$ is*

$$depth_s(x) = \min_p\{t : U(p) = x \text{ in } t \text{ steps and } |p| < K(p) + s\}$$

Li and Vitányi [LV97b] proved the following relating the last two definitions of depth.

**Theorem 3.1.10** *A string $x$ is $(d,b)$-deep ($b$ up to precision $K(d)+O(1)$) if and only if $d$ is the least time needed by a $b$-incompressible ($K(p) \geq |p| - b$) program to print $x$.*

Note that by considering a weighted account of near-minimal programs yields a complexity measure that is robust and machine-independent. This machine-independence results of the ability of the Universal Turing machines to simulate one another. This

means that if an object is shallow on one machine it will not be very deep on another, since we can simulate the second on the first with an constant number of extra bits and an polynomial increase on the running time. Bennett has proved that a fast deterministic processes is unable to transform a shallow object into a deep one, and that fast probabilistic processes are able only with small probability, as it were by accident.

**Definition 3.1.11** *The algorithmic entropy of a string $x$, denoted as $H(x)$ is defined as the least integer greater than $-\log \sum_{p:U(p)=x} 2^{-|p|}$, and the conditional entropy $H(x|y)$ is defined similarly as the least integer greater than $-\log \sum_{p:U(p,y)=x} 2^{-|p|}$*

**Lemma 3.1.12** *There exists constants $c_1$ and $c_2$ such that for any string $x$, if programs running in time $\leq t$ contribute a fraction between $2^{-s}$ and $2^{-s+1}$ of the string's total algorithmic probability, then $x$ has depth at most $t$ at significance level $s + c_1$ and depth at least $t$ at significance level $s - \min\{H(s), H(t)\}$.*

**Theorem 3.1.13 (Slow Growth Law)** *Given a string $x$ and two significance parameters $s_2 > s_1$, a random program generated by coin tossing has probability less than $2^{-(s_2-s_1)+O(1)}$ of transforming $x$ into an excessively deep output, one whose $s_2$-significance depth exceeds the $s_1$-significance depth of $x$ plus the run time of the transforming program plus $O(1)$.*

**Proof.** Let $p$ be a $s_1$ incompressible program computing $x$ in time $depth_{s_1}(x)$. Let

$$Q = \{q : depth_{s_2}(U(q, x)) > time(q, x) + depth_{s_1}(x) + c_1\},$$

where $c_1$ is a constant sufficiently large to cover the time overhead of concatenation and $time(q, x)$ is the running time of $q$ with input $x$. For all $q \in Q$ the program $rpq$, $r$ is a constant prefix restarting the computation, by definition computes deep result $U(q, x)$ in less time than the result own $s_2$-significant depth, so $rpq$ must be compressible by at least $s_2$ bits. If we now consider the algorithmic probability of the string of the form $rpq$ with $q \in Q$ we have

$$\sum_{q \in Q} P(rpq) < \sum_{q \in Q} 2^{-|rpq|+s_2} = 2^{-|r|-|p|+s_2}\mu(Q).$$

As we can recover $p$ from any string of the form $rpq$, the algorithmic probability of $p$ is greater or equal as the sum of the algorithmic probabilities of the strings $\{rpq : q \in Q\}$, i.e.,

$$P(p) > 2^{-|r|-|p|+s_2-O(1)}\mu(Q).$$

Now, by the coding theorem we have

$$\mu(Q) < 2^{-(s_2-s_1)+O(1)}$$

◇

On the other hand , the above prove shows that deep strings can be produced by slow computations using a diagonal argument. First we generate a set of strings quickly computable by small programs, then output the first string not in this set.

Logically deep strings are not easy to identify, but can be constructed by diagonalization in time larger than $2^t$ for depth $t$. Consider the following program:

`Find all` $n$`-bit strings whose algorithmic probability, from computations halting within time` $t$`, is greater than` $2^{-n}$`. Print the first string not in this set.`

It takes time about $t2^t$ to evaluate the algorithmic probability (explicitly simulating all $t$ bits coin tosses sequences), outputting a specific string $\aleph(n,t)$ having a $t$ fast algorithmic probability less than $2^{-n}$. By Lemma 3.1.12 we know that $\aleph(n,t)$ has depth at least $t$ at significance level

$$n - H(t) - \min\{H(n - H(n) - H(t)), H(t)\} - O(1)$$

which is at least $n - H(t) - O(\log n)$.

Note also that the halting sequence must be deep. It has the ability to speed up any slow computation, in particular the program considered above, therefore by the slow growth law, the halting set must itself be deep.

Given a string $x$, its length $n$, and the significance parameter $s$, we can easily encode them in a self-delimiting program of size $n - s + O(\log n)$, we can compute $depth_s(x)$ which by definition must be less than $BB(n - s + O(\log n))$. So the busy beaver is an upper bound on the depth of finite strings.

### 3.1.3 Some new results about Logical Depth

Based on the work of Hartmanis [Har83] we obtain some results using the set of all small depth strings of length $n$. We prove that if the satisfiability of Boolean formulas of low depth can be determined in polynomial time, then there exists an NP-complete set that is not $p$-isomorphic to $SAT$.

**Definition 3.1.14** $LD_s = \{x \in \{0,1\}^n : depth_s(x) \leq \log n\}$, *i.e., $LD_s$ is the set of all low depth strings.*

**Theorem 3.1.15** *If $LD_s \cap SAT \subseteq A_0 \subset SAT$ and $A_0 \in \mathrm{P}$, then $SAT - A_0$ is an* NP*-complete set which is not p-isomorphic to $SAT$.*

**Proof.** $SAT \leq_M^P SAT - A_0$ reducing any element in $A_0$ to a fixed satisfiable formula not in $A_0$, and all the other string to themselves.
Now suppose that $SAT$ and $SAT - A_0$ where $p$-isomorphic, then there exist an isomorphism $f$ in P that maps shallow strings in $A_0 \subset SAT$ onto deeper strings in $SAT - A_0$, which contradicts Theorem 3.1.13. Thus $SAT$ is not $p$-isomorphic to $SAT - A_0$. ◇

There at least $2^n(1 - \frac{1}{2^{\log n}}) + 1$ strings $\log n$ incompressible, so the set $LD_s$ has at least $2^n(1 - \frac{1}{2^{\log n}}) + 1$ strings.

**Theorem 3.1.16** *If $Tally \cap \mathrm{NP} \subseteq \mathrm{P}^{LD_s}$, then* NE = E.

**Proof.** Consider the oracle Turing machine $M$ such that if $M$ makes query $q$ and $q \in LD_s$, $C(q) \simeq \log n$, then $C^{poly}(q) \simeq \log n$. So we can generate all such $q$ in polynomial time and $Tally \cap \mathrm{NP} \subseteq \mathrm{P} \Rightarrow \mathrm{NE} = \mathrm{E}$. ◇

**Corollary 3.1.17** *If* $\mathrm{NP} \subseteq \mathrm{P}^{LD^{t_1,t_2}}$, *then* NE = E.

## 3.2 Time-$t$ Depth and Shallow Sets

In this section we discuss shallow sets, sets containing little nonrandom information. We will show that computable sets reducible to shallow sets must have small circuits.

In particular if NP-complete sets reduce to shallow sets then the polynomial-time hierarchy collapses.

First we give a simple notion of depth based on a time complexity function $t$. We argue that computational depth should be based on the difference of two Kolmogorov complexity related measure. So the next definition is as general as possible, being the difference between a time bounded notion of Kolmogorov complexity and the unbounded associated notion. As we have seen before there exist more than one proposal for an individual object complexity, so by instantiating each one of them we get a different notion of computational depth.

**Definition 3.2.1** *Let $t$ be a time-bound. The time-$t$ depth of a string $x$, $D^t(x)$, is*

$$D^t(x) = C^t(x) - C(x).$$

To properly define shallow sets we need to use Definition 3.2.1 for sub-linear time bounds, on the initial segments of the characteristic sequence the set $A$, i.e.,

$$A(\epsilon)A(0)A(1)A(00)\ldots$$

We give a definition for $C^t$ for sub-linear time functions $t$ by allowing the universal machine $U$ random access to the description $r$ of the string $x$ (denoted as oracle access $U^r$, in order to avoid some notation overlap we stress that in this section whenever we use this notation we are referring to oracle access and not to time bound) and requiring only that each bit of $x$ be generated in the allotted time. As we only allow sub-linear time we stress that we do not have time enough even to read the input, so in we allow an oracle access otherwise this measure would not be useful.

**Definition 3.2.2 (Sub-linear time-bounded Kolmogorov complexity)** *Let $t$ be a time-bound and $x$ a string.*

$$\begin{aligned} C^t(x) \;=\; & \min_{p,r}\{|p| + |r| : U^r(p,i) \text{ outputs } x_i \\ & \text{in } t(|x|) \text{ steps for all } 1 \leq i \leq |x|\}. \end{aligned}$$

This definition is essentially equivalent to Definition 2.3.22 for super-linear $t$.

We now define shallow strings and shallow sets.

**Definition 3.2.3 (Shallow strings)** *Fix a constant $k$. The string $x$ is* $k$-shallow *if*

$$D^{\log^k}(x) \leq \log^k |x|.$$

In the proof of our main result Theorem 3.2.5 we will be interested in the characteristic sequences available to some Turing machine running in time $n^j$ on some input of length $n$. The initial segment of the characteristic sequence up to length $n^j$ has length $N = 2^{n^j+1} - 1$. In that case, $\log^k N$ is approximately $n^{jk}$.

We now define shallow sets.

**Definition 3.2.4 (Shallow sets)** *A set $A$ is* shallow *if there exists a $k$ such that almost every initial characteristic sequence of $A$ is $k$-shallow.*

Every sparse set is shallow. In fact, every set that is polynomial-time reducible to a sparse set is shallow. Random sets are also shallow: A randomly chosen set is shallow with probability one.

Despite the fact that most sets are shallow, we now show that these sets have very limited computational power.

**Theorem 3.2.5** *If we have sets $A$ and $B$, $A$ shallow, $B$ computable and $B \in \mathrm{P}^A$ then $B$ is in* P/poly.

To prove Theorem 3.2.5 we need the following result of Nisan and Wigderson [NW94]:

**Lemma 3.2.6 (Nisan-Wigderson)** *For any fixed nonnegative integer $d$, there exists a family of generators $\{G_0, G_1, \ldots\}$ with the following properties:*

- *$G_v$ maps strings of length $u$ polynomial in $\log v$ to strings of length $v$.*
- *For any circuit $D$ of depth $d$ and size $v$, we have*

$$\left| \Pr_{\rho \in \{0,1\}^v}[D(\rho)] - \Pr_{\sigma \in \{0,1\}^u}[D(G_v(\sigma))] \right| < 1/v.$$

- *Each output bit of $G_v$ is computable in time polynomial in $\log v$.*

*Proof of Theorem 3.2.5.* By assumption there is some Turing machine $M$ running in time $n^j$ for some $j$, such that $B = L(M^A)$, and all sufficiently long initial segments of the characteristic sequence of $A$ are $k$-shallow for some nonnegative integer $k$.

Let $a_i$ be the $i$th bit of the characteristic sequence of $A$. Fix some input length $n$. Let $z = a_1 \ldots a_{2^{n^j+1}-1}$ be the characteristic sequence of $A$ up to strings of length $n^j$. Let $N = |z| = 2^{n^j+1} - 1$. We have that $C^{n^{jk}}(z) - C(z) \leq n^{jk}$. Let $\ell \doteq C^{n^{jk}}(z)$, which gives us $C(z) \geq \ell - n^{jk}$. Note that $\ell \leq |z| < 2^{n^j+1}$.

By Definition 3.2.2 there must be a $p$ and $r$ such that $|p| + |r| = \ell$ and $U^r(p, i)$ outputs $a_i$ in time $n^{jk}$ for each $i$, $1 \leq i \leq N$. Note that $C(z) \leq C(\langle p, r\rangle) + O(\log n)$.

Now consider the set $T$ consisting of all pairs $(q, s)$ such that

- $|q| + |s| = \ell$,
- For $1 \leq i \leq N$, $U^s(q, i)$ halts in time $n^{jk}$ and outputs some value $f_i$.
- For each string $y$ of length $n$, $y$ is in $B$ if and only if $M^F(y)$ accepts where $F$ is the oracle whose characteristic sequence is $f_1 f_2 \ldots f_N 000 \ldots$.

The set $T$ has some nice properties:

- $(p, r)$ is in $T$.
- $T$ can be computed by a constant depth circuit of size $2^{n^{O(1)}}$, namely as follows. For each string $y$ of length $n$, we have to verify that the oracle machine $M$ accepts $y$ when its oracle queries, say about the value of $f_i$, are answered by running $U^s(q, i)$ for $n^{jk}$ steps. Since $M$ runs in time $n^j$, for any fixed $y$, this process can be viewed as a computation running in time $n^j \cdot n^{jk}$ with random access to $(q, s)$. Such a computation can be expressed as an OR of $2^{n^{j(k+1)}}$ AND's of size $n^{j(k+1)}$ each over the input $(q, s)$. AND'ing all these circuits together for all $y$'s of length $n$ yields a depth 3 circuit of size $2^n \cdot 2^{n^{j(k+1)}} \cdot n^{j(k+1)}$ deciding $T$. Call this circuit $D$.
- For each pair $(q, s)$ in $T$, $C(\langle q, s\rangle) \leq \log |T| + O(\log n)$ since $B$ is computable.

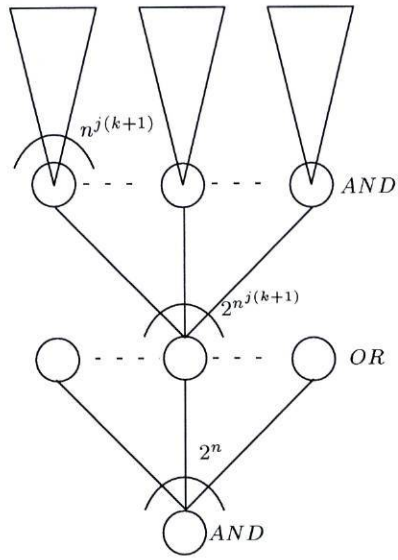


Figure 3.1: Checking whether a pair is good is easy.

By the third item we have

$$\begin{aligned}\log|T| &\geq C(\langle p,r\rangle) - O(\log n)\\ &\geq C(z) - O(\log n)\\ &\geq \ell - n^{jk} - O(\log n).\end{aligned}$$

This gives us $|T| \geq 2^\ell/2^{n^c}$ for some constant $c$.

Let $v$ be the max of $2^{n^c}$ and the size of the circuit $D$ describing $T$. Let $G_v$ be the Nisan-Wigderson generator from Lemma 3.2.6. We have

$$|\Pr_{\rho\in\{0,1\}^v}[D(\rho)] - \Pr_{\sigma\in\{0,1\}^u}[D(G_v(\sigma))]| < 1/v.$$

Since $D$ picks $(q,s)$ uniformly from the initial bits of $\{0,1\}^v$ we have

$$\Pr_{\rho\in\{0,1\}^v}[D(\rho)] = \Pr_{|q|+|s|=\ell}[(q,s)\in T] \geq |T|/2^\ell \geq 1/v.$$

So we have

$$\Pr_{\sigma\in\{0,1\}^u}[D(G_v(\sigma))] > 0.$$

In particular, there is some $\sigma$ such that $D(G_v(\sigma))$ is true. This $\sigma$ has length polynomial in $\log v$ which is polynomial in $n$. We let this $\sigma$, $v$, $|q|$ and $|s|$ be our advice. From the advice we can efficiently compute every bit of a pair $(q,s)$ in $T$ which we can use to determine membership in $B$ on strings of length $n$. ◇

Karp and Lipton [KL80] show that if NP-complete languages have polynomial-size circuits then the polynomial-time hierarchy collapses to the second level. This gives us the following corollary.

**Corollary 3.2.7** *If any* NP*-complete language is Turing-reducible to a shallow set then the polynomial-time hierarchy collapses to* $\Sigma_2^p$.

Balcázar, Díaz and Gabarró [BDG86] showed the following characterization of $PSPACE/\text{poly}$.

**Theorem 3.2.8** $A \in PSPACE/\text{poly}$ *if and only if for every* $n$ *the characteristic sequence of the set* $A$ *of strings up to length* $n$*, has logarithmic Kolmogorov complexity by machines using polynomial space.*

We can use shallow sets to prove a similar result to characterize the computable sets in P/poly. Hartmanis argued that his approach could not be used to characterize P/poly because of the time needed for writing the output. We feel Definition 3.2.2 handles these issues well.

**Corollary 3.2.9** *A computable set* $C$ *is in* P/poly *if and only if* $C$ *is shallow.*

## 3.3 Distinguishing Computational Depth

In this section we introduce another variant of computational depth based on the difference between time bounded Kolmogorov complexity and time bounded distinguishing complexity. We prove a close analog of Bennett's slow growth law and also show how to find in quasi-polynomial probabilistic time a satisfying assignment to any satisfiable Boolean formula for which a significant fraction of the satisfying assignments has logarithmic depth.

**Definition 3.3.1** *Let* $x, y$ *be strings, and* $t_1, t_2$ *be time-bounds. The* $(t_1, t_2)$-distinguishing computational depth *of* $x$ *given* $y$ *is*

$$D^{t_1,t_2}(x|y) = C^{t_1}(x|y) - CD^{t_2}(x|y).$$

It is clear that the distinguishing computational depth is always nonnegative. The exact difference between $C^t(x|y)$ and $CD^t(x|y)$ is not known. It is conceivable that both measures are always very close, in which case the notion of distinguishing depth would become trivial. However, this is unlikely because Fortnow and Kummer [FK96] showed that in that case the promise problem $(1SAT, SAT)$ can be solved in polynomial time.

We now start working towards the analog of Bennett's slow growth law for honest efficiently computable functions with few inverses. A function $f$ is honest if for some polynomial $p$, $p(|f(x)|) \geq |x|$ for all $x$.

We will need the following lemma.

**Lemma 3.3.2** *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable function that is at most $m$-to-1. For every polynomial $p_1$ there exists a polynomial $p_2$ such that for any string $x$ of length $n$ and any string $y$*

$$CD^{p_2}(x|y) \leq CD^{p_1}(f(x)|y) + 2\log m + O(\log n).$$

*If $f$ is a one-to-one (injective) function we have*

$$CD^{p_2}(x|y) \leq CD^{p_1}(f(x)|y) + O(1).$$

**Proof.** Let $p'$ be the program that distinguishes $f(x)$ given $y$. We create a program that on input $\langle z, y\rangle$ accepts only if $z = x$ as follows:

1. Simulate $p'$ on $\langle f(z), y\rangle$ and reject if $p'$ rejects. Otherwise we have $f(z) = f(x)$. If $f$ is one-to-one we have $x = z$ and $p'$ just accepts.
2. If $m > 1$, run a program that recognizes $x$ among the at most $m$ other strings that map to $f(x)$.

The first step takes polynomial time. If $f$ is one-to-one we immediately get Lemma 3.3.2.

For the second step we can apply Theorem 2.3.29 to the set $A \doteq \{u : f(u) = f(x)\}$ given both $y$ and $f(x)$. Note that $|A| \leq m$. Since $f$ is polynomial-time computable, and we are given $f(x)$, we can simulate the queries to $A$ in time polynomial in $n$. Therefore, we have that

$$CD^p(x|y, f(x)) \leq 2\log m + c\log n$$

for any sufficiently large polynomial $p$ and constant $c$.

All together we get that for any large enough polynomial $p_2$

$$\begin{aligned} CD^{p_2}(x|y) &\leq CD^{p_1}(f(x)|y) \\ &\quad + CD^{p}(x|y, f(x)) + O(\log n) \\ &\leq CD^{p_1}(f(x)|y) \\ &\quad + 2\log m + O(\log n). \end{aligned}$$

◇

The analog to Bennett's slow growth law for logical depth reads as follows for distinguishing computational depth.

**Theorem 3.3.3** *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time computable honest function that is at most $m$ to 1. For all polynomials $p_1, p_2$ there exist polynomials $q_1, q_2$ such that for any string $x$ of length $n$ and any string $y$*

$$D^{q_1,p_2}(f(x)|y) \leq D^{p_1,q_2}(x|y) + 2\log m + O(\log n).$$

*If $f$ is one-to-one we have*

$$D^{q_1,p_2}(f(x)|y) \leq D^{p_1,q_2}(x|y) + O(1).$$

**Proof.** In order to produce $f(x)$, we can first produce $x$ and then run a program for $f$ on $x$. Since $f$ is polynomial-time computable and honest, we have that for any polynomial $p_1$ there exists a polynomial $q_1$ such that

$$C^{q_1}(f(x)|y) \leq C^{p_1}(x|y) + O(1). \tag{3.1}$$

Lemma 3.3.2 tells us that for any polynomial $p_2$ there exists a polynomial $q_2$ such that

$$CD^{p_2}(f(x)|y) \geq CD^{q_2}(x|y) - 2\log m - O(\log n) \tag{3.2}$$

for $m > 2$ and

$$CD^{p_2}(f(x)|y) \geq CD^{q_2}(x|y) - O(1) \tag{3.3}$$

if $f$ is one-to-one.

Subtracting (3.2) or (3.3) from (3.1) as appropriate finishes the proof of the theorem. ◇

We next prove that if the depth of a nonnegligible fraction of the satisfying assignments of a Boolean formula is small then we can find a satisfying assignment in quasi-polynomial probabilistic time.

**Theorem 3.3.4** *For all functions $q(n) = 2^{\log^d n}$, there exist a polynomial $p$ and a probabilistic quasi-polynomial-time algorithm that given any satisfiable Boolean formula $\phi$ of size $n$ such that at least a $1/q(n)$ fraction of the satisfying assignments $x$ to $\phi$ have*

$$D^{q,p}(x|\phi) \leq \log^d n,$$

*the algorithm outputs an assignment of $\phi$ with high probability.*

**Proof.** Fix a satisfiable Boolean formula $\phi$ of length $n$ and let $A$ denote the set of satisfying assignments of $\phi$.

From Theorem 2.3.32 we know that there exists a polynomial $p$ and constant $c$ such that, for all but a $\frac{1}{2q(n)}$ fraction of the satisfying assignments $x$ to $\phi$, we have

$$CD^p(x|\phi) \leq \log|A| + \log^c n.$$

By hypothesis, at least a fraction $\frac{1}{2q(n)}$ of the $x$ in $A$ must also satisfy

$$C^q(x|\phi) \leq CD^p(x|\phi) + \log^d n,$$

so we have

$$C^q(x|\phi) \leq \log|A| + \log^b n$$

for some constant $b$.

Now we randomly chose a program of length at most $k + \log^b(n)$, for every $1 \leq k \leq n$. Giving these programs as input to the universal Turing machine $U$ we can produce in quasi-polynomial time a satisfying assignment of $\phi$ with probability at least

$$\frac{\frac{|A|}{2q(n)}}{2^{\log|A|+\log^b(n)+1}} \geq 2^{-\log^a n}$$

for some positive constant $a$. Repeating this procedure a quasi-polynomial number of times will produce a satisfying assignment of $\phi$ with high probability. ◇

## 3.4 Basic Computational Depth

By considering $Ct(x) - C(x)$ we get a variation of Bennett's logical depth.

**Definition 3.4.1** *For any string $x$ the basic computational depth of $x$ is*

$$\begin{aligned} bcd(x) &= Ct(x) - C(x) \\ &= \min_p\{\log(t) + |p| - C(x) : U(p) \text{ outputs } x \text{ in } t \text{ steps}\}. \end{aligned}$$

Basic computational depth has the advantage over logical depth that we do not need a significance level parameter. In fact, we are adding to (the logarithm of) the running time of the program a penalty $|p| - C(x)$ which can be viewed as a significance level.

Logically deep strings are not easy to identify, but can be constructed by diagonalization in time larger than $2^t$ for depth $t$ [Ben88]. We prove that there are an exponential number of strings with large basic computational depth. The result holds for Bennett's notion of logical depth as well.

**Theorem 3.4.2** *There exists a constant $c$ such that for any $0 < \epsilon < 1$ there are at least $2^{\epsilon n}$ strings $x$ of length $n$ satisfying*

$$bcd(x) \geq (1 - \epsilon)n - c \log n.$$

**Proof.**

Consider the set $A$ consisting of all strings $x$ of length $n$ for which there exists a program $p$ of length less than $n$ such that $U(p)$ outputs $x$ in at most $2^n$ steps.

Let $B$ denote $\{0,1\}^n \setminus A$ and $C$ the lexicographically first $2^{\epsilon n}$ strings in $B$. The set $C$ exists and we know that for every $x \in C$, $C(x) \leq \epsilon n + O(\log n)$.

So for all $x \in C$ every program $p$ of size less than $n$ such that $U(p)$ outputs $x$ must run for at least $2^n$ steps. It follows that for any $x \in C$, $bcd(x) \geq (1 - \epsilon)n - c \log n$ for some constant $c$. ◇

# 4

# Sophistication vs Computational Depth

The information contained in an individual finite object is commonly measured by its Kolmogorov complexity. We can partition the information into two parts: the part accounting for the useful regularity present in the object and the part accounting for the remaining accidental information. The useful information is called a "statistic" of the data. Kolmogorov suggested that the useful information be represented by a finite set of which the object is a typical member, so that the two-part description of the finite set together with the index of the object in that set is as parsimonious as the shortest one-part description. The finite set statistic models the regularity present in the object (since it is a typical element of the set). This approach has been generalized to computable probability mass functions. The combined theory has been developed in detail in [GTV01] and called "Algorithmic Statistics."

The most general way to proceed is perhaps to express the useful information as a recursive function. The resulting measure has been called the "sophistication" of the object in [Kop87, Kop88, KA91]. In this Chapter we show that the results in [Kop87] are wrong, however we consider the concept of sophistication as interesting and propose a mathematical investigation of the notion for finite strings. Technically, we develop the sophistication based on prefix Turing machines, rather than on a variety of monotonic Turing machines as in the cited papers.

In this Section 4.1 we describe the concepts of sophistication, Kolmogorov minimal

sufficient statistic and algorithmic statistics. In Section 4.2 we present some new results on sophistication; namely we show that sophistication is a generalization of the Algorithmic Statistics [GTV01], and that there are strings $x \in \Sigma^n$ with sophistication close to $n$. We also give a brief analysis of the results in [Kop87]. Motivated by the previous analysis in Section 4.3 we introduce a new variant of computational depth, namely Busy Beaver Computational Depth and show some basic features of this measure. In Section 4.4 we discuss the notion of sophistication and present a new variant of sophistication Coarse Sophistication proving the following interesting result: for all $x \in \Sigma^*$, given $x$ and $O(\log n)$ bits, we can solve the halting problem for all programs $q$ of size smaller than $\frac{soph(x)}{2} - 2\log n$. Finally, in Section 4.5 we prove the equivalence between Busy Beaver Computational Depth and Coarse Sophistication.

## 4.1 Sophistication vs Algorithmic Statistics

The shortest effective description of an object $x$ can be expressed in terms of a *two part code:* the first part is a description of an appropriate Turing machine and the second part is the program that interpreted by the Turing machine reconstructs $x$. The essence of the Invariance Theorem is as follows: For the fixed reference universal Turing machine $U$ the length of the shortest program to compute $x$ is $\min\{|p| : U(p) = x\}$. From the definition it follows that

$$K(x) = \min_{T,p}\{|T_n| + |p| : T_n(p) = x\} \pm 1$$

note that $U$ expects inputs of the format $< n, p >= \underbrace{11...1}_{|n|\times}0p$ and by convention $U(0p) = U(p)$. This provides an alternative definition of prefix Kolmogorov complexity. The above expression for Kolmogorov complexity can be rewritten as

$$K(x) = \min_{i}\{K(T_i) + K(x|T_i) : i \in \mathbf{N}\} + O(1)$$

that emphasizes the two-part code nature of Kolmogorov complexity: it uses the regularity of $x$ to compress it as much as possible.

**Example 4.1.1** *Consider*

$$x = 10101010101010101010101010$$

*We can encode $x$ by a small Turing machine that computes $x$ from the program "13". Intuitively, the Turing machine part of the code squeezes out the* regularities *in $x$. What is left are irregularities, or* random aspects, *of $x$ relative to that Turing machine. The minimal-length two-part code squeezes out regularity only insofar as the reduction in the length of the description of random aspects is greater than the increase in the regularity description.*

We can interpret $K(x)$ as the shortest length of a two part code for $x$, one describing the Turing machine (*model* or *valuable* information in $x$) for the *regular* aspects of $x$ and the second describing the *irregular* aspects of $x$ (*useless* information in $x$) in the form of a program to be interpreted by $T$.

The "right model" is a Turing machine $T$ among the ones that reach the minimum description length

$$\min\{K(T_i) + K(x|T_i) : i \in \mathbf{N}\} + O(1)$$

This $T$ embodies the amount of useful information contained in $x$. The main question now is: from the Turing machines that satisfy this requirement which one should we select? The problem is how to separate a shortest program for $x$ into parts $pq$ such that $p$ represents an appropriate $T$. This idea has spawned the theoretical notion of Kolmogorov minimal sufficient statistic [She83, Cov85], algorithmic statistic [GTV01] and sophistication [Kop87, Kop88, KA91].

The following definition was introduced by Kolmogorov in a talk at the Information Theory Symposium, Talin, Estonia, in 1974 see [Cov85].

**Definition 4.1.2 (Kolmogorov)** *The Kolmogorov structure function $H_k : \Sigma^n \to \mathbf{N}$ is defined as*

$$H_k(x) = \min_{p:|p|\leq k} \log |S|,$$

*where the minimum is taken over all subsets $S \subseteq \Sigma^n$, such that $x \in S$, for all $y \in S$ $U(p) = y$ and $|p| \leq k$.*

$H_k$ is the log of the size of the smallest set containing $x$ over all sets specifiable by a program of $k$ or fewer bits. Of special interest is the value

$$K^*(x) = \min\{k : H_k(x) + k = K(x)\}.$$

A program for $x$ can be written in two stages:

1. `Use` $p$ `to print the indicator function for` $S$.
2. `The desired string is the` $ith$ `sequence in a lexicographic ordering of the elements of this set.`

This program has length $|p| + \log|S| + O(1)$, and $K^*(x)$ is the length of the shortest program $p$ for which this two-stage description is as short as the best one stage description. Note that $x$ must be maximally random with respect to $S$, otherwise the two stage description could be improved, contradicting the minimality of $K(x)$.

Although all finite sets are recursive, there are different ways to specify the set. Gács *et al.* [GTV01] only consider ways that have in common a method of recursively enumerating the elements of the finite set one by one, and differ in knowledge of its size. A representation of a finite set $S$ is *explicit* if the size $|S|$ of the finite set can be computed from it, and it is *implicit* if the log size $= \log|S| + O(1)$ can be computed from it.

**Example 4.1.3** *In [GTV01] the set $S^k$ of strings whose elements have complexity not exceeding $k$ is investigated. This set can be represented implicitly by a program of size $K(k)$, but can be represented explicitly only by a program of size $k$.*

Such representations are useful in two-stage encodings where one stage of the code consists of an index in $S$ of length $= \log|S| + O(1)$. In the implicit case we know, within an additive constant, how long an index of an element in the set is. The worst case of representation format $X$, a recursively enumerable representation where nothing is known about the size of the finite set, would lead to indices of unknown length.

Gács *et al.* [GTV01] generalize the model class from finite sets to probability distributions. Instead of finite sets the models are computable probability density functions. "Computable" means that there is a Turing machine $T_P$ that computes approximations to the value of $P$ for every argument. The complexity $K(P)$ of a computable partial function $P$ is defined by

$$K(P) = \min_i \{K(i) : \text{Turing machine } T_i \text{ computes } P\}.$$

The Kolmogorov function now becomes

$$K(x \mid P) = -\log P(x) + O(1),$$

and

$$K(x) = K(P) - \log P(x) + O(1).$$

The complexities involved are crucially dependent on what we mean by "computation" of $P(x)$, that is, on the requirements on the format in which the output is to be represented. Turing machines can compute rational numbers [LV97b]: if a Turing machine $T$ computes $T(x)$, then we interpret the output as a pair of natural numbers, $T(x) = \langle p, q \rangle$, according to a standard pairing function. Then, the rational value computed by $T$ is by definition $p/q$. Gács *et al.* distinguished between explicit and implicit description of $P$ as follows:

- It is *implicit* if there are positive constants $c_1$ and $c_2$ such that the Turing machine $T$ computing $P$ halts with rational value $T(x)$ with $c_1 P(x) < T(x) < c_2 P(x)$. Hence $-\log T(x) = -\log P(x) + O(1)$.
- It is *explicit* if the Turing machine $T$ computing $P$, given $x$ and a tolerance $\epsilon$ halts with rational value $P(x) - \epsilon < T(x) < P(x) + \epsilon$.

The implicit and explicit descriptions of uniform distributions with $P(x) = 1/|S|$ for all $x \in S$ and $P(x) = 0$ otherwise, are as follows: An implicit (explicit) description of $P$ is identical with an implicit (explicit) description of $S$, up to a short fixed program which indicates which of the two is intended, so that $K(P(x)) = K(S) + O(1)$ for $P(x) > 0$ (equivalently, $x \in S$).

In fact, since $K(x \mid P) < K(x) + O(1)$, it limits the size of $P(x)$ to $\Omega(2^{-k})$. The shortest program $P^*$ from which a probability density function $P$ can be computed is an *algorithmic statistic* for $x$ if and only if

$$K(x \mid P^*) = -\log P(x) + O(1). \tag{4.1}$$

There are two natural measures of suitability of such a statistic. We might prefer either the simplest distribution, or the largest distribution, as corresponding to the most likely structure 'explaining' $x$. The singleton probability distribution $P(x) = 1$,

while certainly a statistic for $x$, would indeed be considered a poor explanation. Both measures relate to the optimality of a two-stage description of $x$ using $P$:

$$\begin{aligned} K(x) \leq K(x,P) &= K(P) + K(x \mid P^*) + O(1) \\ &< K(P) - \log P(x) + O(1), \end{aligned} \tag{4.2}$$

Call a distribution $P$ with positive probability $P(x)$ optimal if

$$K(x) = K(P) - \log P(x) + O(1), \tag{4.3}$$

Depending on whether $P$ is understood as implicitly described or explicitly described, definition splits into implicit and explicit optimality. The shortest program for an optimal computable probability distribution is an *algorithmic sufficient statistic* for $x$, and a shortest algorithmic sufficient statistic is called an *algorithmic minimal sufficient statistic* for $x$.

Expressing the useful information as a recursive function Koppel [Kop87, Kop88, KA91] introduced the concept of "sophistication" of an object based on process (monotonic) complexity defined by Schnorr [Sch73]. A more stable definition of monotonic complexity was introduced by Levin [Lev73a].

Schnorr's definition on of monotonic complexity is based on the notion of a monotonic function. A function $f : \Sigma^* \to \Sigma^*$ is monotonic if $x \leq y$ ($x$ is a prefix of $y$) implies that $f(x) \leq f(y)$ for all $x$ and $y$.

**Definition 4.1.4 ([Sch73])** *Let $\phi$ be a monotonic function,*

$$Km_\phi(x) = \min_p\{|p| : \phi(p) = x\}$$

Schnorr also proved an invariance theorem for this measure, so we can use $Km(x)$.

**Definition 4.1.5 (Sophistication)** *The sophistication of an infinite string $\alpha$ is,*

$$\begin{aligned} soph(\alpha) \;=\; \min_p\{|p| : &\; p \text{ is total, exists } c \text{ for all } n \text{ exists } d_n \\ &\text{such that } |p| + |d_n| \leq Km(x) \text{ and } d_{n-1} \leq d_n\} \end{aligned}$$

Note that sophistication does not obey an invariance law, i.e., it is not possible to have a total universal function. This is interesting, because in order to prove an invariance

law we need to consider a universal non-total machine. We just require that the programs $p$ to be total.

In 1982 at a seminar in the Moscow State University, see [V'y99], Kolmogorov raised the question if "absolutely non-random" (or absolutely non-stochastic) objects exist. Those rare objects for which the simplest models that summarize their relevant information are at least as complex as the objects themselves.

**Definition 4.1.6 (Kolmogorov)** *Let $\alpha$ and $\beta$ be natural numbers. A string $x \in \Sigma^n$ is called $(\alpha, \beta)$-stochastic if there exists a finite set $S$ such that $x \in S$, $K(S) \leq \alpha$ and $K(x) \geq \log |S| - \beta$.*

The first inequality (if $\alpha$ is not too large) means that $S$ is sufficiently simple. The second (if $\beta$ is not too large) means that $x$ is an "typical" element in $S$. Indeed, if $x$ had some feature peculiar to only a very small part $Q$ of $S$, then these could be used for a simple description of $x$ by determining its ordinal number in a list of all the elements in $Q$, which could require $\log |Q|$ bits, i.e., many fewer than $\log |A|$.

The following theorem, proved by Shen [She83], is part of the answer for a corresponding problem posed by Kolmogorov about the existence of "absolutely non-random" strings.

**Theorem 4.1.7 ([She83])**

1. *There exists a constant $c$ such that, for any $n$ and any $\alpha$ and $\beta$ with $\alpha \geq \log n + c$ and $\alpha + \beta \geq n + 4 \log n + c$, all the numbers from 0 to $2^n - 1$ are $(\alpha, \beta)$-stochastic.*

2. *There exists a constant $c$ such that, for any $n$ and any $\alpha$ and $\beta$ with $2\alpha + \beta < n - 6 \log n - c$ not all the numbers from 0 to $2^n - 1$ are $(\alpha, \beta)$-stochastic.*

Gács *et al.* [GTV01] improved Shen's result and proved that for every $n$ there are objects of length $n$ with complexity $K(x|n) \approx n$ such that every explicit algorithmic sufficient statistic for $x$ has complexity about $n$ ($\{x\}$ is such a statistic).

## 4.2 Some Results on Sophistication

In [Kop87] Koppel claimed that depth and sophistication for all infinite strings are equivalent. However the proof is wrong and also uses a different definition of depth imposing totality in the functions defining depth.

**Claim 4.2.1** *There is a constant $c$ such that for all describable infinite sequence $\alpha$*

$$|soph(\alpha) - depth(\alpha)| \leq c.$$

In fact it is possible to have some discrepancy between depth and sophistication.

**Example 4.2.2** *Let $\phi$ be a universal partial recursive function. Define $\chi = \chi_1\chi_2\chi_3\cdots$ the characteristic sequence of the diagonal halting set $\{x : \phi_x(x)\downarrow\}$*

$$\chi_i = \begin{cases} 1 & if \quad \phi(i,i) = 0 \\ 0 & if \quad \phi(i,i) \neq 0 \text{ or } \phi(i,i)\uparrow \end{cases}$$

*By the Barzdins Lemma (see [LV97b]) $\log n \leq C(\chi_{1:n}|n) \leq \log n + c$, so $soph(\chi_{1:n}) \leq \log n + c$, but $depth(\chi_{1:n})$ must be very high since $\chi_{1:n}$ has the ability to speed up any slow computation.*

We view the notion of sophistication as genuinely interesting and propose a mathematical investigation of the notion for finite strings. We start by refine Koppel definition of sophistication preserving the main properties. Instead of monotonic Kolmogorov complexity we use prefix-free Kolmogorov complexity, and we will only consider it for finite strings.

**Definition 4.2.3** *Let $c$ be a constant, $x \in \Sigma^n$ and $U$ the universal reference Turing machine*

$$soph_c(x) = \min\{|p| : p \text{ is total, exists } d\ [U(p,d) = x \text{ and } |p| + |d| \leq K(x) + c]\}.$$

We prove that the sophistication is smaller than the Kolmogorov minimal sufficient statistic plus a constant $l$ that depends on the constant used in the definition of the sophistication.

**Theorem 4.2.4** *For all constant $c > 0$ there is a constant $l > 0$ such that for all $x \in \{0,1\}^n$,*

$$soph_c(x) - l \leq \textit{length minimal sufficient statistic for } x.$$

**Proof.** Let $p$ be the program that defines $soph_c(x)$, $p^*$ the Kolmogorov minimal sufficient statistic for $x$ and $k = |K(|d|)| \leq \log\log n$. We want to prove that ($|p^*| \geq |p| - l$) Suppose that $|p^*| < |p| - l$. By definition we know that exists $S^* = \{y : U(p^*) = y\}$, $x \in S^*$ and $\log|S^*| + |p^*| \leq K(x) + c$.
Let $p'$ the program that is the same as $p^*$ except that any data string $1 \leq d \leq |S^*|$ is used to get the $d^{th}$ element of $S^*$ in lexicographic order, and all other data $d$ is treated as if $d = 0$. Then $p'$ is total, $U(p', i) = x_i$ where $x_i$ is the $i^{th}$ element in lexicographic order in $S^*$ and exists $c_1$ such that

$$|p'| \leq |p^*| + c_1 < |p| - l + c_1.$$

So if we chose $l$ such that $l > c_1$ then we can use the total program $p'$ and $\log|S^*|$ bits to define the sophistication of $x$, so $soph(x) = |p'| =< |p|$, which contradicts $soph_c(x) = |p|$. ◇

Regarding the question raised by Kolmogorov about the existence of "absolutely non-random" (or absolutely non-stochastic) objects we note that we can reformulate the question using sophistication and ask if exists strings $x \in \Sigma^n$ such that the sophistication is close to $n$. As we have seen before Shen [She83] has partially answer this question and Gács *et al.* [GTV01] has fully answered it for explicit algorithmic sufficient statistic. Next we present a "simple" proof of the existence of strings $x \in \Sigma^n$ such that the sophistication is close to $n - c\log n$, the proof is done using a diagonalization argument.

**Theorem 4.2.5** *For every $n \in I\!N$ there is a string $x \in \{0,1\}^n$ such that $soph_c(x) > n - 2\log n - 2c$.*

**Proof.** For all $p$ such that $|p| \leq n - 2\log n - 2c$ we define

$$r_p = \begin{cases} 0 \text{ if } \exists d : |d| < n - |p| - c \text{ such that } U(p,d) \uparrow \\ \max\limits_{d:|d|<n-|p|-c} \text{running time of } U(p,d) \end{cases}$$

Let $S = \max r_p$. Given $n$ and $p$ that maximizes $r_p$ we can compute $S$. Consider

$$U = \{x : \exists p, d \ |p| \leq n - 2\log n - 2c, \ |d| \leq n - |p| - c \text{ s.t. } U^t(p,d) = x, \ t \leq S\}.$$

Let $z$ be the least, in the lexicographic order, element in $\{0,1\}^n$ such that $z \notin U$. Such $z$ exists since $\forall x \in U, K(x) \leq |p| + |d| \leq |p| + n - |p| - c = n - c$ and by a simple counting argument there must exist at least $2^n(1 - \frac{1}{2^c})$ strings $z \in \{0,1\}^n$ with $K(z) \geq n - c$. By construction we know that $K(z) \leq K(p) + 2\log n \leq n - 2c$.
Assume that $soph_c(z)$ is small, i.e., $soph_c(z) \leq n - 2\log n - 2c$; then, by definition

$$\exists p^*, d^* : p^* \text{ is total, } |p^*| \leq n - 2\log n - 2c \text{ and } |p^*| + |d^*| \leq K(z) + c$$

but then we have that $|p^*| \leq n - 2\log n - 2c$ and $|d^*| = K(z) + c - |p^*| \leq n - |p^*| - c$ so $U(p^*, d^*)$ runs in time $\leq S$, i.e., $z \in U$. But by construction $z \notin U$, so $soph_c(z) > n - 2\log n - 2c$. ◇

We can get a sharper result, similar to the result proved by Gács *et al.* [GTV01], using also conditional Kolmogorov complexity.

**Corollary 4.2.6** *If we replace $K(x)$ by $K(x|n)$ in the definition of sophistication then*

$$\exists x \in \{0,1\}^n : soph_c(x) \geq n - c$$

Based on this last result we turn our attention to the concept of randomness with respect to a set. Kolmogorov wanted to use algorithmic complexity to eliminate the need for a direct interpretation of probabilities. We believe that Kolmogorov idea was that the randomness of a finite sequence $x \in S$ manifests itself in the absence of regularities in $x$, which can be interpreted as the absence of a description of $x$ much shorter than a typical element in $S$.

**Definition 4.2.7** *The randomness deficiency of $x$ relative to $S$ is defined as*

$$\delta(x|A) = \log|A| - K(x|A).$$

*It follows that $\delta(x|A) \geq -c$ for some constant $c$ independent of $x$.*

The randomness deficiency estimates the difference in length between two descriptions of $x$ by means of the set $S$: the standard and shortest description. Note that by this definition the randomness deficiency is "almost positive", i.e., some strings may have negative randomness deficiency relative to some set.

**Definition 4.2.8** *For all $x \in \Sigma^n$ we define $rand_c(x) = K(x) - soph_c(x) + c$*

We have just proved that there are some $x \in \Sigma^n$ such that $soph_c(x) \geq n - c$, so $n - c \leq K(x|n) \leq n + c_1$ which is equivalent to say that

$$\delta(x|\Sigma^n) \leq c$$

so we can conclude that these $x$ are random.

However we know by construction that $rand_c(x) \leq 2c + c_1 = c_2$, i.e., this object has lots of structure but it is somehow "hidden". Based on this result we propose that we should consider the following definition for randomness deficiency.

**Definition 4.2.9** *The randomness deficiency of $x \in \Sigma^n$ is defined as*

$$\delta(x|A) = \log|A| - \mathrm{rand}_c(x).$$

This new definition not only consider this highly sophisticated strings but it is also always positive.

## 4.3 Busy Beaver Computational Depth

In [Kop87] Koppel tried to prove that depth and sophistication for all infinite strings are equivalent. As we have seen before this can not happen. In fact this would be an really unexpected result as we where comparing two completely different things, time (depth) that can grow unbounded and program length (sophistication) that is upper bounded by the length of the string. In the attempted proof, Koppel used a different definition of depth imposing totality in the functions defining depth and using an Busy Beaver argument.
However, we believe that imposing totality in the notion of depth in order to compare it with sophistication is not fair. Without this restriction we get a notion close to basic computational depth.

The main drawback of basic computational depth is the fact that it is only suitable for strings whose program runs in time at most exponential in the length of the string. However, we could be interested in programs with running time bigger than

exponential. This motivate the definition of busy beaver computational depth, preserving the intuition of basic computational depth and capturing all possible running times. Besides using properly the busy beaver function, we can upper bound the computational depth by the size of the string.

**Definition 4.3.1** *Let $x$ be a string, the busy beaver computational depth of $x$ is given by*

$$depth_{bb}(x) = \min\{|p| - C(x) + k : U(p) = x \text{ in } t \text{ steps and } k \leq BB^{-1}(t)\}$$

The intuition behind this definition is that instead of considering a significance level as Bennett did, we incorporate this in the formulae. The term $|p| - C(x)$ is a penalty measuring how far away we are from the minimal program. It is important to note that, instead of being using the running time $t$, we are using the inverse Busy Beaver of the running time, i.e., this way we are scaling down this measure to program length. Using this new measure we are able to study strings whose minimal programs can have running time bigger than exponential, as in basic computational depth, however we lose some "sensibility" as now $depth_{bb}$ is upper bounded by $\frac{n}{2}$.

**Theorem 4.3.2** *There is a constant $c$ such that for all $x \in \{0,1\}^n$, $depth_{bb}(x) \leq \frac{n}{2}+c$.*

**Proof.**
If $C(x) \leq \frac{n}{2}$, considering the minimum program producing $x$ we have $depth_{bb}(x) \leq \frac{n}{2} + c$.
If $C(x) > \frac{n}{2}$, considering the $print(x)$ program we have $depth_{bb}(x) \leq \frac{n}{2} + c$.
$\diamond$

Now we prove that there are strings for which this upper bound is tight.

**Theorem 4.3.3** *For all $n \in \mathbb{N}$ exists $x \in \{0,1\}^n : depth_{bb}(x) > \frac{n}{2} - 4\log n$*

**Proof.** Let $U$ be the set of all $x \in \Sigma^n$ such that

$$\text{exists } p, k,\ |p| \leq \frac{n}{2} - 2\log n,\ k \leq \frac{n}{2} - 2\log n - |p|,\ U(p) = x \text{ in } t \text{ steps and } t \leq BB(k)$$

Consider $z$ the least, in the lexicographic order, element in $\Sigma^n$ such that $z \notin U$. Such $z$ exists since for all $x \in U, C(x) < |p| + 2\log n = \frac{n}{2}$. We can approximate $BB$ from

below so $U$ is r.e. and by construction we know that the size of $U$ is smaller than $|p| + 2\log n$ so $C(z) < \frac{n}{2} + 2\log n$.
Assume $depth_{bb}(z) \leq \frac{n}{2} - 4\log n$, then by definition exists a $p$ and $k$ such that

$$k + |p| - C(z) \leq \frac{n}{2} - 4\log n \text{ and } U(p) = x \text{ in } t \text{ steps and } t \leq BB(k)$$

i.e.

$$k \leq \frac{n}{2} - 2\log n - |p| \text{ and } U(p) = x \text{ in } t \text{ steps and } t \leq BB(k)$$

so $z \in U$. But by construction $z \notin U$, so $depth_{bb}(z) > \frac{n}{2} - 4\log n$. ◇

## 4.4 Coarse Sophistication

In this section we introduce a new variant of the notion of sophistication. We think that Koppel definition of sophistication is not stable, so we decided to look for a more robust variant, capturing the spirit of the initial concept of sophistication. We incorporate the constant $c$ that appears in the sophistication formulae as a penalty, just as we have done for basic computational depth and Busy Beaver computational depth.

**Definition 4.4.1** *The coarse sophistication of a string $x$ is*

$$csoph(x) = \min\{2|p| + |d| - C(x) : U(p,d) = x \text{ and } p \text{ is total.}\}$$

We can see $|p| + |d| - C(x)$ as a penalty, or how far away we are from the minimal program, and $|p|$ as the length of the program defining the sophistication. The following interpretation of Coarse Sophistication was given by Paul Vitányi, in fact he has also suggested the name Coarse Sophistication.

Starting from the usual notion of sophistication, $soph(x) = |p|$ summarizes the regularity in $x$ and $d$ summarizes the accidental features. However, $p$ summarizes *all* the regularity in $x$, at all levels. For example, consider a painting, one level of regularity is the image, another level is the brush strokes, and so on. To consider the regularity of *only* the image we have to account for the *level* and hence require a description that is larger than $K(x)$, say of length $K(x) + \Delta C$ where $\Delta C$ is the "compressibility" of

the new description. If we describe the regularity of only the image by the program $r$ then clearly $|r| < |p| = soph(x)$.

Now we can possibly argue that the most significant regularity, or the optimal regularity, the best level of coarseness, $r$ is the one such that $\Delta(p) = |p| - |r|$ exceeds $\Delta C$ as much as possible, that is, we want to maximize $\Delta p - \Delta C$ (the savings in bits in terms of total program minus the added compressibility).

What is considered in coarse sophistication is $|r| + \Delta C$ such that $\Delta(p) = |p| - |r|$ and $\Delta p - \Delta C$ is maximized.

So this is the length of the total program at the optimal level of coarseness plus the extra compressible bits required in the description. Note that reducing bits of the total program corresponds to coarse graining the description universe and adding compressible bits can perhaps be viewed as indexing the object in the coarse grain.

As it happened for Busy Beaver computational depth, in Coarse Sophistication we lose some "sensibility" as now $csoph(x)$ is upper bounded by $\frac{n}{2}$.

**Theorem 4.4.2** *There is a constant $c$ such that for all $x \in \{0,1\}^n$, $csoph(x) \leq \frac{n}{2} + c$.*

**Proof.**
If $C(x) \leq \frac{n}{2}$ then by definition $csoph(x) \leq \frac{n}{2} + c$.
If $C(x) > \frac{n}{2}$ then considering the $print(x)$ program we have $csoph(x) \leq \frac{n}{2} + c$.
$\diamond$

Now we prove that there are strings for which this upper bound is tight.

**Theorem 4.4.3** *For all $n \in \mathbb{N}$ exists $x \in \{0,1\}^n$ such that $csoph(x) > \frac{n}{2} - 2\log n$.*

**Proof.** For all $p$ such that $|p| \leq n - 2\log n$ we define

$$r_p = \begin{cases} 0 \text{ if exists } d : |d| < n - |p| \text{ such that } U(p,d) \uparrow \\ \max\limits_{d:|d|<n-|p|} \text{running time of } U(p,d) \end{cases}$$

Let $S = \max r_p$. Given $n$ and $p$ that maximizes $r_p$ we can compute $S$. Consider

$$U = \{x : \text{ exists } p, d \ |p| \leq \frac{n}{2} - 2\log n, \ |d| \leq n - |p| - 4\log n \text{ s.t. } U^t(p,d) = x, \ t \leq S\}.$$

Let $z$ be the least, in the lexicographic order, element in $\{0,1\}^n$ such that $z \notin U$. Such $z$ exists since for all $x \in U, C(x) < |p| + |d| + 4\log n \leq |p| + 4\log n - |p| - 4\log n = n$ and by a simple counting argument there exists random strings. By construction we know that

$$C(z) \leq C(p) + 2\log n \leq \frac{n}{2}.$$

Assume that $csoph(z)$ is small, i.e., $csoph(z) \leq \frac{n}{2} - 2\log n$; the program $p^*$ which defines the sophistication is such that

$$|p^*| \leq \frac{n}{2} - 2\log n$$

and there exists $d^*$ such that

$$2|p^*| + |d^*| - C(z) \leq \frac{n}{2} - 2\log n$$

but then we have that

$$|d^*| \leq \frac{n}{2} - 4\log n + C(z) - 2|p^*| \leq n - 4\log n - 2|p^*|$$

so $U(p^*, d^*)$ runs in time $\leq S$, i.e., $z \in U$.
But by construction $z \notin U$, so

$$csoph(z) > \frac{n}{2} - 2\log n.$$

◇

We now study the computational power of the notion of coarse sophistication. We prove an interesting result result relating the coarse sophistication with the halting problem: knowing the sophistication of a string and some extra $\log n$ bits we can solve the halting problem for all programs of length smaller than $\frac{csoph(x)}{2} - 2\log n$. This is a very strong result which suggests a connection between the sophistication and the computational depth: by the slaw growth law, the halting sequence is deep (it can speed up any slow computation), thus strings with high sophistication must be deep.

**Theorem 4.4.4** *For all $x \in \Sigma^*$, given $x$ and $O(\log n)$ bits we can solve the halting problem for all programs $q$ such that $|q| < \frac{csoph(x)}{2} - 2\log n$.*

**Proof.** Find the minimum program $p$ such that $U(p) = x$, and consider $S$ its running time. Suppose that there is some $q$ such that $|q| < \frac{csoph(x)}{2} - \log n$ and $U(q)$ converges

in time $v > S$. Consider the program $w$ such that $U(w,p)$ first computes $v$ and then simulates $U(p)$ for $v$ steps, producing $x$. Now $w$ is total and there is a constant $c$ such that $|w| = |q| + c$, and

$$\begin{aligned} csoph(x) &\leq 2|q| + 2c + |p| + 2\log n - C(x) \\ &\leq csoph(x) - 2\log n + 2c + |p| - C(x) \\ &< csoph(x) - 2\log n + 2c \end{aligned}$$

so such an input can not exist. ◇

## 4.5 Coarse Sophistication vs Busy Beaver Computational Depth

In this section we prove the main results in this chapter the equivalence of busy beaver computational depth and coarse sophistication. Thus this concepts has an inherent relevance that transcends the realm of pure mathematical abstraction:

*In mathematics the fact that quite different formalizations of concepts turn out to be equivalent is often interpreted as saying that the captured notion has an inherent relevance that transcends the realm of pure mathematical abstraction.*

*(Li and Vitányi [LV97b])*

**Theorem 4.5.1** *For all $x \in \Sigma^*$, $|csoph(x) - depth_{bb}(x)| \leq O(\log n)$.*

**Proof.**

We start proving that $depth_{bb}(x) \leq csoph(x) + O(\log n)$.
For all $d$ such that $|d| \leq n$ let $t$ be the maximum running time of $U(p_s, d)$, then using $p_d$ and $\log n$ bits to describe $d$ we get $t \leq BB(|p_s| + O(\log n))$. So we get an upper bound on the sophistication of $x$

$$\begin{aligned} depth(x) &\leq |p_s| + |d| - C(x) + |p_s| + O(\log n) \\ &\leq csoph(x) + O(\log n) \end{aligned}$$

Now we prove that $csoph(x) \leq depth_{bb}(x) + O(\log n)$.
We denote the running time of a program $p$ by $rt(p)$. Let $q$ be the first program of length $k$ whose running time immediately follows the running time of $p_d$, i.e.

$$|q| = k,\ rt(q) > rt(p_d) \text{ and for all } u : |u| = k,\ rt(u) > rt(p_d) \Rightarrow rt(q) < rt(u).$$

Consider the set

$$A = \{v : |v| = |p_d|,\ rt(v) < rt(q) \text{ and for all } u : |u| = k,\ rt(u) > rt(v) \Rightarrow rt(u) > rt(q)\}.$$

Given $q, n, |p|$ and $k$, $A$ is recursive, since $rt(q)$ is used as the time limit for the running time of all programs. By symmetry of information we have

$$C(v|q) \leq C(q|v) + C(v) - C(q) + O(\log n).$$

But, given $v$ we can use its running time to get $q$, because $q$ is the first program of length $k$ whose running time immediately follows the running time of $p_d$, so $C(q|v) \leq O(\log n)$ and $C(v|q) \leq |p_d| - k + O(\log n)$.
As $A$ is recursive we have $|A| \leq 2^{|p_d|-k+O(\log n)}$ and we can point $p_d$ with its index $i$ in $A$. Note that for every $p \in A$, $rt(p) < rt(q)$, i.e., it halts. Now consider the code of the machine that with input $< q, n, |p|, k >$ and $i$ constructs the set $A$ and picks $p_d$ that given as input to the universal Turing machine produces $x$. Then

$$\begin{aligned} csoph(x) &\leq 2|q| + O(\log n) + |i| - C(x) \\ &\leq 2k + O(\log n) + |p| - k - C(x) \\ &\leq k + |p| - C(x) + O(\log n) \\ &\leq depth(x) + O(\log n) \end{aligned}$$

◇

# 5

# Average Case Complexity vs Computational Depth

The complexity of a problem is usually defined in terms of the worst case complexity of algorithms. However despite having a bad worst case behavior, many algorithms are frequently used in practice because they are efficient on average. It seems that the instances which cause the bad worst case complexity are rare in many practical applications. Thus, in some cases, the average case complexity of a problem is a more significant measure than its worst case complexity. The general question of average case complexity was addressed for the first time by Levin [Lev86]. He introduced the concept of average polynomial-time for measuring "easiness" on average and the notion of average case NP-completeness for measuring "hardness" on average. Levin then showed that a certain tiling problem is average case NP-complete if each parameter of an instance is randomly selected.

In this Chapter we show an interesting relation between computational depth and Levin's average case complexity. The type of depth we consider, $t$-time bounded depth of $x$, defined as the difference between $K^t(x)$ and $K(x)$, is a variant of time-bounded computational depth.

Li and Vitányi [LV97a] studied the universal distribution, $\mathbf{m}$ introduced by Solomonoff [Sol64], which assigns $\frac{1}{2^{-K(x)}}$ weight to each string $x$. They showed that when the inputs to any algorithm are distributed according to the universal distribution, the algorithm's average case complexity is of the same order of magnitude as its worst case complexity.

We now rephrase this connection in terms of average polynomial-time.

**Fact (Li-Vitányi)** *The running time $t$ of a machine is polynomial on average with respect to the (unbounded) universal distribution if and only if $t$ itself is bounded by a polynomial.*

Since the universal distribution is not even recursive, this result fits poorly with the traditional average case complexity analysis, where the distribution must be "simple" (usually considering polynomial-time computable or samplable distributions). Therefore, unless we can prove some kind of time bound version of Li and Vitányi's result, the two subjects seem quite unrelated.

In Section 5.3.2 we show the following result.

**Main Result** *The running time $t$ of a machine is polynomial on average with respect to the time-bounded universal distribution if and only if $t$ is bounded by $2^{depth_t(x)}.\text{poly}(|x|)$*

The main result can be viewed as a generalization of Li and Vitányi's result. Indeed, as $t$ goes unbounded depth approaches 0 and $\mathbf{m}^t$, the time bounded version of $\mathbf{m}$, approaches $\mathbf{m}$. As an interesting corollary we get that, if the running time $t$ of a machine is exponentially bounded in the depth, then $t$ is polynomial on average with respect to all computable distributions of certain time bound.

**Corollary** *If the running time $T$ of a machine $M$ is bounded by $2^{depth_t(x)}.\text{poly}(—\text{x}—)$ then $L(M)$ is in Average polynomial-time with respect to any distribution computable in time $t/n$.*

Based on the connection between pseudo-random generators and time-bounded Kolmogorov complexity, Schuler [Sch99] showed that if a polynomial-time computable distribution dominating $\mathbf{m}^t$ exists, then no polynomially secure pseudo-random generators exists. So it is unlikely that there are polynomial-time computable distributions dominating universal distributions.

Motivated by this relation and the the main result, we also show that there are samplable distributions dominating the time-bounded universal distributions.

In Section 5.1 we give some definitions and results from average case computational complexity. In Section 5.2 we present the known results about the universal distribution and time bound universal distribution. In Section 5.3 we present our main

results.

## 5.1 Average Case Computational Complexity Theory

The complexity of a problem is usually defined in terms of the worst case complexity of algorithms. However it is unlikely that a problem needs to be solved on all instances. The problem is that we do not know *a priori* which instances will occur as input as we could code the hard instances in the algorithm. The most general and *naive* approach is to assign to each instance a probability with which we believe the instance will in practice occur as input. The average case complexity can now be defined as the expected value of the complexity of the algorithm, taken over all instances. Often, when specific distributions of the inputs are assumed, the average case time complexity of specific algorithms shows a significant improvement over the worst case complexity. Instances which occur with high probability should be solved quickly. The hope is to show that even for intractable problems, hard instances occur only with small probability. Hence, some algorithm should run efficiently on average.

The expected polynomial time is an obvious choice for the concept of average case efficiency of a function $f : \Sigma^* \to \mathbf{N}$. In particular, such a definition might require that

$$\sum_{|x|=n} \mu'_n(x) \times f(x) \leq O(n^k)$$

for all $n \in \mathbf{N}$ and some constant $k$, where the normalized distribution is $\mu'_n(x) = \frac{\mu'(x)}{\sum_{|x|=n} \mu'(x)}$. This requires that the expected value of $f$ over inputs of length $n$ be bounded by a polynomial in $n$.

However, this definition has two main drawbacks: it is machine dependent (depends on the manner in which the instances are encoded) and it is not closed under composition with a polynomial.

Levin [Lev86] introduced a model independent, closed under composition and encoding independent definition of polynomial on average.

**Definition 5.1.1** *A semi-distribution function $\mu : \{0,1\}^* \to [0,1]$ is a non-decreasing function from strings to the unit interval $[0,1]$. A distribution function $\mu : 0,1^* \to [0,1]$*

*is a non-decreasing function from strings to the unit interval* $[0,1]$ *which converges to one. The density function associated with the distribution function* $\mu$ *is denoted* $\mu'$ *and defined by* $\mu'(0) = \mu(0)$ *and* $\mu'(x) = \mu(x) - \mu(x-1)$ *for every* $x > 0$. *Clearly,* $\mu(x) = \sum_{y \leq x} \mu'(y)$.

Usually semi-distributions can be normalized to distributions without changing the complexity.

**Definition 5.1.2** *A function* $f : \Sigma^* \to \mathbf{N}$ *is polynomial on* $\mu$*-average if there exists an* $\epsilon > 0$ *such that* $\sum_x f^\epsilon(x)|x|^{-1}\mu(x) < \infty$ *(converges).*

Impagliazzo [Imp95] observed that this definition is equivalent to taking the average on instances up to length $n$ since, when $n$ is sufficiently large, $\mu(|x| \leq n)$ is greater than a fix positive constant.

**Lemma 5.1.3** *A function* $f$ *is polynomial on* $\mu$*-average if and only if there exists* $\epsilon > 0$ *and* $c > 0$ *such that for all* $n$ $\sum_{x:|x|\leq n} f^\epsilon(x)\mu_{\leq n}(x) \leq cn$, *where* $\mu_{\leq n}$ *is the conditional distribution of* $\mu$ *on* $\{x : |x| \leq n\}$.

From the definition it follows that any polynomial is polynomial on $\mu$-average for any $\mu$. It is easy to show that if the functions $f$ and $g$ are polynomial on $\mu$-average, then the functions $f + g$, $f \times g$ (it suffices to note that $f \times g \leq f^2 + g^2$), and $f^k$ for some constant $k$ are also polynomial on $\mu$-average.

A problem is solvable in average polynomial-time with respect to distribution $\mu$ if it can be solved by a deterministic algorithm whose running time is polynomial on $\mu$-average. A problem associated with a probability distribution is called a distributional problem.

**Definition 5.1.4** *A distributional decision (search) problem is a pair* $(L, \mu)$ (resp. $(S, \mu)$), *where* $L : \Sigma^* :\to \{0,1\}$ (resp. $S \subseteq \Sigma^* \times \Sigma^*$) *and* $\mu$ *is a distribution function.*

### 5.1.1 Distributions

Average case analysis in general is sensitive to the choice of distribution, so we will study in more detail the behaviors and properties of individual distributions. In this

section we explain the basic concepts of distribution and density functions.

We study two different categories of distributions: *polynomial-time computable* distributions considered by Levin [Lev86] and *polynomial-time samplable* distributions considered by Ben-David *et al.* [BDCGL92].

We use the notion of *domination* for comparing distributions.

**Definition 5.1.5** *Let $\mu$ and $\nu$ be semi-distributions. Let $t$ and $\mathcal{M}$ be a function and a set of functions from $\Sigma^*$ to $R^+$, respectively. The semi-distribution $\nu$ $t$-dominates $\mu$ if $t(x)\nu'(x) \geq \mu'(x)$ for all $x \in \Sigma^*$, and $\nu$ $\mathcal{M}$-dominates $\mu$ if there exists a function $t' \in \mathcal{M}$ such that $\nu$ $t'$-dominates $\mu$.*

**Definition 5.1.6** *Let $\mathcal{M}$ be a class of semi-distributions. A semi-distribution $\mu$ is universal, or maximal, for $\mathcal{M}$ if $\mu \in \mathcal{M}$, and for all $\nu \in \mathcal{M}$, there exists a constant $c_\nu$ such that for all $x \in \Sigma^*$, we have $c_\nu \mu(x) \geq \nu(x)$.*

Note that if $\mu$ is a universal semi-distribution for $\mathcal{M}$ then it multiplicatively dominates each $\nu \in \mathcal{M}$. The class of all semi-distributions has no universal semi-distributions, the same holds for the class of recursive semi-distributions.

### Polynomial-time Computable Distributions

In average case complexity if we allow arbitrary distributions then average case complexity classes take the form of traditional worst-case complexity classes [Gol88]. So it is important to restrict attention to distributions which are in some sense *simple*. Usually simple distributions are identified with the P-computable ones. We will start by defining the class of distributions such that for all $x$, the value of $\mu(x)$ is expressed by some polynomial-time Turing machine, the class of strictly polynomial-time distributions.

**Definition 5.1.7 (strict-P-computable)** *A distribution $\mu$ is in the class strict-*P*-computable if there is a deterministic polynomial time Turing machine that on input $x$ outputs the binary expansion of $\mu(x)$ (the running time is polynomial in $|x|$).*

Usually we relax this definition by considering a Turing machine, $M$, that approximates $\mu(x)$, i.e., $|\mu(x) - M(x, 0^i)| \leq 2^{-i}$ for all $i \in N$.

**Definition 5.1.8** *A probability distribution function $\mu$ on $\{0,1\}^*$ is* polynomial-time computable, *if there is a deterministic Turing machine that on every input $x$ and a positive integer $i$, runs in time poly$(|x| + i)$, and outputs a fraction $y$ such that $|\mu^*(x) - y| \leq 2^{-i}$.*

**Lemma 5.1.9 ([BG91])** *1. If $f$ and $g$ are polynomial time computable functions from $\Sigma^*$ to the real interval $[0,1]$, then $f+g$,$f-g$ and $f \times g$ are polynomial-time computable.*

*2. Let $f$ be a monotone function from $\Sigma^*$ to the real interval $[0,1]$ and let $A(x, 1^k)$ witness the polynomial time computability of $f$. There exists a witness $B(x, 1^k)$ to the polynomial-time computability of $f$ such that, for every $k$, $B$ is monotone in $x$.*

Note that, by the previous Lemma, if a distribution is $t$-time computable, then the density function is also $t$-time computable. However Blass showed that the converse is not true unless P = NP (see [Gur91]).

**Theorem 5.1.10 ([Gur91])** *If* P $\neq$ NP *then, there exits a density function which is computable in polynomial-time, but its associated distribution is not* P*-computable.*

**Proof.** Consider the set $A$ in NP $-$ P such that

$$A = \{x : \text{ exists } y\,(|y| = |x| \text{ and } < x, y > \in B)\}$$

for some set $B \in$ P.

Let

$$\nu'(x) = \begin{cases} 0 & \text{if } |x| \text{ is odd;} \\ \frac{2^{-2n}}{(n+1)(n+2)} & \textit{otherwise.} \end{cases}$$

it is easy to see that $\nu$ is in P-comp. Now consider the following distribution $\mu$:

$$\mu'(z) = \begin{cases} \nu'(xy) & \text{if } z = x0y, |x| = |y|, \text{ and } < x, y > \in B; \\ \nu'(xy) - \mu'(x1y) & \text{if } z = x1y \text{ and } |x| = |y|; \\ 0 & \textit{otherwise.} \end{cases}$$

We need to prove that $\sum_z \mu'(z) = 1$.

$$\sum_z \mu'(z) = \sum_{(x,y):|x|=|y|} (\mu'(x0y) + \mu'(x1y)) = \sum_{(x,y):|x|=|y|} \nu'(xy) = 1$$

By definition we know that $x \in A$ if and only if $\mu(x10^{|x|}) - \mu(x0^{|x|+1}) \neq 0$. So if $\mu \in$ P-comp, then $A \in$ P. Therefore, $\mu$ is not in P-comp. ◇

In the sequel, when we say that a distribution $\mu$ is polynomial-time computable we assume that both $\mu$ and $\mu'$ are polynomial time computable.

Note however that polynomial-time computability of $\mu$ does not guarantee the polynomial time computability of the $k$-digit of $\mu(x)$. For, let $M$ be a Turing machine that computes a binary function $b(x)$ from $\Sigma^*$ to $\{0,1\}$ such that the sets $\{x : b(x) = 0\}$, $\{x : b(x) = 1\}$ are recursively inseparable. Let $T(x)$ be the running time of $M(x)$; $T(x)$ is infinite if $M(x)$ does not halt. If $M(x)$ halts, let

$$\mu(x) = 0.0(01)^{T(x)}1 \text{ and } \nu(x) = 0.0(10)^{T(x)}b(x).$$

Otherwise, let

$$\mu(x) = 0.0(01)^{\infty} \text{ and } \nu(x) = 0.0(10)^{\infty}.$$

$\mu$, $\nu$ are polynomial-time computable. Let $f = \mu + \nu$, if $b(x) = 0$ then $f(x) = 0.0...$ and if $b(x) = 1$, then $f(x) = 0.1$. Thus, computing the first digit of $f$ would separate the inseparable sets.

Levin conjectured that any "natural" probability distribution is either polynomial time computable or else is dominated by one that is. Gurevich [Gur91] showed that all polynomial time distributions are bounded above by nicely behaved polynomial time computable distributions.

**Theorem 5.1.11** *Let $\mu$ be a polynomial time computable distribution. Then there is a distribution $\nu$ such that, for all $x$, $\nu(x) > 0$, $\nu'(x)$ has at most $4 + 2|x|$ digits and*

$\mu'(x) < 4\nu'(x)$. *Moreover, there is a deterministic algorithm that on input* $x$ *outputs* $\nu'(x)$ *in time polynomial in* $|x|$.

**Proof.** Since $\mu$ is polynomial-time computable, there is a polynomial-time computable finite binary fraction $b(x)$ such that, for all $x$, $|\mu(x) - b(x)| < \frac{d_x}{2}$, where $d_x = 2^{-2|x|}$. Round $b(x)$ down to $2|x|+1$, add $\frac{d_x}{2}$ if the last digit is 1. This produces a binary fraction $B(x)$ with at most $2|x|$ digits, and $|\mu(x) - B(x)| < d_x$. Define $\nu'$ by $\nu'(x) = \frac{1}{4}(B(x) - B(x^-) + 3d_x)$, where $x^-$ is the immediate predecessor of $x$. Then $\nu'(x)$ has at most $4 + 2|x|$ digits and

$$\nu(x) = \frac{1}{4}(B(x) + 3\sum_{y \le x} d_y) = \frac{1}{4}(B(x) + 3\sum_{n=1}^{|x|} 2^{-n}) \to 1 (as |x| \to \infty).$$

It follows that

$$4\nu'(x) > (\mu(x) - d_x) - (\mu(x^-) + d_x) + 2d_x = \mu'(x)$$

◇

Although it is impossible to select strings with equal chance from an infinite sample space, strings of the same length can be selected with equal likelihood.

**Definition 5.1.12** *A polynomial time computable distribution* $\mu$ *on* $\Sigma^+$ *is called universal if, for all* $x$, $\mu(x) = \rho(|x|)2^{-|x|}$, *where* $\sum_n \rho(n) = 1$ *and there is a polynomial* $p$ *such that, for all but finitely many* $n$, $\rho(n) \ge \frac{1}{p(n)}$.

The second condition in the definition guarantees that almost every length gets a "fair" degree of probability weight. Levin [Lev86] used $n^{-2}$ for $\rho(n)$ for notational convenience, and $n^{-2}2^{-|x|}$ is often referred as the default uniform distribution.

Levin [Lev86] proved that distributions that are polynomial time computable can be dominated, with respect to polynomial-time computable function, by uniform distributions. Gurevich latter gave a different and easier proof, based on this proof Wang and Belanger [WB95] showed that if the distribution is not too small, it will also dominate the same uniform distribution within a constant factor.

**Lemma 5.1.13 (Distribution Controlling Lemma)** *Let* $\mu$ *be a polynomial-time computable distribution.*

1. *There exists a total, one-to-one, polynomial-time computable and invertible function* $g : \Sigma^+ \to \Sigma^+$ *such that, for all* $x$, $\mu'(x) < 2^{-|g(x)|+2}$.

2. *Moreover, if there exists a polynomial* $p$ *such that, for all* $x$, $\mu(x) > 2^{-p(x)}$, *then there is a total, one-to-one, polynomial time computable and invertible function* $f : \Sigma^+ \to \Sigma^+$ *such that, for all* $x$, $4 \times 2^{-|f(x)|} \leq \mu'(x) \leq 20 \times 2^{-|f(x)|}$.

**Proof.**

1. Let $\mu$ be polynomial time computable and $\mu(x) > 2^{-p(x)}$. Consider the distribution $\nu$ constructed in Theorem 5.1.11. Let

$$g(x) = \min\{y : \nu(x^-) < 0.y1 \leq \nu(x)\}$$

$g$ is total, one-to-one, polynomial-time computable and invertible. Now by minimality of $g(x)$, we have $0.w < \nu(x^-)$ and $\nu(x) \leq 0.w^+$, where $w = g(x)$. So, $\nu'(x) = \nu(x) - \nu(x^-) < 0.w^+ - 0.w = 2^{-|w|}$, i.e., $\nu'(x) < 2^{-|g(x)|}$. Therefore, $\mu'(x) < 2^{-|g(x)|+2}$.

2. Consider the function $g$ constructed in (1). As $\frac{\mu'(x)}{4}$ is polynomial-time computable, take a deterministic Turing machine $M$ such that $|\frac{\mu'(x)}{4} - M(x, 0^i)| < 2^{-i}$ and let $N(x) = M(x, 0^{p(|x|)+4})$. Let $d(x)$ be the position of the leftmost digit 1 in the binary fraction of $N(x)$, i.e., $2^{-d(x)} \leq N(x) < 2^{-d(x)+1}$. Now defining

$$f(x) = g(x)10^{d(x)-|g(x)|}$$

clearly $f$ is one-to-one, polynomial-time computable and invertible. Now we will show that $d(x) \leq p(|x|) + 3$. Assume otherwise. By definition, we know that $N(x) < 2^{-d(x)+1} \leq 2^{-p(|x|)-3}$, and then $\frac{\mu'(x)}{4} < N(x) + 2^{p(|x|)-4} < 2^{-p(|x|)-2}$, contradiction. So, we have

$$\frac{\mu'(x)}{4} < N(x) + 2^{p(|x|)-4} \leq 2^{-d(x)-1} + 2^{-d(x)-1} = 5 \times 2^{-d(x)-1} = 5 \times 2^{-|f(x)|}.$$

Now, let $d'(x)$ be the position of the leftmost digit 1 in the binary fraction of $\frac{\mu'(x)}{4}$, i.e., $2^{-d(x)} \leq \frac{\mu'(x)}{4} < 2 \times 2^{-d(x)}$. Assume that $d'(x) > d(x)+1$, since $N(x) \geq 2^{-d(x)}$ and $\frac{\mu'(x)}{4} < 2^{-d(x)+1}$, we have $N(x) - \frac{\mu'(x)}{4} > 2^{-d(x)} - 2^{-d(x)+1} \geq 2^{-d(x)-1}$. Then we get a contradiction, $N(x) - \frac{\mu'(x)}{4} \geq 2^{-p(|x|)-4}$. Using the fact that $d'(x) \leq d(x)+1$, we conclude that $\frac{\mu'(x)}{4} \geq 2^{-d'(x)} \geq 2^{-d(x)-1} \geq 2^{-|f(x)|}$.

◇

### Polynomial-time Samplable Distributions

The most controversial definition in the average case complexity theory is the association of the class of "simple" distributions with P-computable, which may seem too restricting. Ben-David *et al.* in [BDCGL92] introduced a wider family of natural distributions, P-samplable, consisting of distributions that can be sampled (or generated) by randomized algorithms, with no input, working in time polynomial in the length of the sample generated. They proved that under "modest" cryptographic assumptions, there are P-samplable distributions that are "very far" from any P-computable distribution.

**Definition 5.1.14** *A distribution $\mu$ is in the class* P*-samplable if there exists a polynomial $p$ and a probabilistic algorithm $A$ that outputs the string $x$ with probability $\mu'(x)$ within $p(|x|)$ steps.*

Note that elements in a P-samplable distribution are generated in time polynomial in their length.

**Theorem 5.1.15** *Every* P*-computable distribution is also* P*-samplable.*

**Proof.** The sampling algorithm $A$ picks a truncated real $\rho$ in $[0,1]$ (the length of expansion depends on the following search) uniformly at random. It then finds, via binary search, and queries to $\mu$, the unique string $x \in \{0,1\}^*$ satisfying $\mu(x^-) < \rho \leq \mu(x)$. It outputs $x$ if $|\rho| \leq -\log(\mu(x) - \mu(x^-))$. Let $S$ be the set of such $\rho$ such that no string in $S$ is a prefix of a different string in $S$. Then the probability of sampling $x$ is equal to $\sum_{\rho \in S} 2^{-|\rho|} = \mu(x)$. ◇

As for polynomial-time distributions, following the ideas in [Yam97] and in order to cope with real valued distributions, we use an approximation scheme and give a generalized definition of polynomial-time samplability. To distinguish this generalized version from the one proposed by Ben-David *et al.* [BDCGL92] Yamakami [Yam97] called theirs P-samplable strict-P-samplable.

**Definition 5.1.16** *A distribution $\mu$ is in the class* P*-samplable if there exists a polynomial $p$ and a randomized Turing machine $M$, called sampling machine or generator,*

*such that*

$$|\mu'(x) - Pr[M \text{ on input } 0^i \text{ produces } x \text{ and halts within time } poly(|x|, i)]| \leq 2^{-i}$$

*for all* $x$ *and* $i \in \mathbf{N}$. *We say that* $M$ *samples* $\mu$ *if* $M$ *satisfies this last condition.*

Ben-David *et al.* [BDCGL92] showed that all P-samplable distributions can be effectively "enumerated" in a certain way to construct a universal P-samplable distribution. They do not enumerate all sampling algorithms running in polynomial time; instead, they enumerate all Turing machines modifying them to stop in time polynomial in the output. This is done by augmenting the machine so it pads its outputs once entering the original halt sate. The padding is long enough to make the machine run in the required time. It is important to state that the output of the original machine remains unchanged. Note that such an enumeration is not known for P-computable functions.

**Theorem 5.1.17 ([BDCGL92])** *There exists an effective enumeration of all strictly* P*-samplable distributions. In particular, for each* $k > 0$, *there is an effective enumeration of all strictly* $O(n^k)$*-time samplable distributions.*

**Proof.** Let $\{M_i\}_{i \in \mathbf{N}}$ be an effective enumeration of all randomized Turing machines, and $\{p_i\}_{i \in \mathbf{N}}$ and effective enumeration of all polynomials with positive integer coefficients such that $p_i(x) \geq z$ for all $z$. For each pair $(i, j)$ of natural numbers, consider the $i$th machine $M_i$ and the $j$th polynomial $p_j$, and modify the machine as follows:

Sampling Algorithm for $M'_{<i,j>}$
**begin**
**input** $\lambda$
simulate $M_i$ on input $\lambda$
let $x$ be the output of the machine $M_i$ and
let $t$ be the running time of $M_i$.
**if** $t \leq p_j(|x|)$ **then output** $x$ **else output** $x0^{t-|x|}$
**end**

Suppose that $M_i$ outputs $x$. The running time of $M'_{<i,j>}$ is $O(|x|+t+1)$. If $t \leq p_j(|x|)$, the running time of $M'_{<i,j>}$ is $O(p_j(|x|))$ because $p_j(n) \geq n$; otherwise, it is $O(t)$.

Overall, the running time of $M'_{<i,j>}$ is at most $cn$ steps in the length of its output. Now, we can easily check that all P-samplable distributions appear in this enumeration. Now, let $\mu_1, \mu_2, ...$ be an enumeration of the distributions generated by (modified) sampling machines. Define a universal distribution

$$\mu'_U(x) = \sum_{i=1}^{\infty} \frac{\mu'_i(x)}{i^2}$$

Clearly, $\mu_U$ is P-samplable, first select $i$ with probability $\frac{1}{i^2}$ and next sample $\mu_i$. $\diamond$

Every P-computable distribution is also P-samplable. Ben-David *et al.* [BDCGL92] proved that the converse is unlikely. Namely,

**Theorem 5.1.18** *If one-way functions exists, then there is a* P*-samplable distribution* $\mu$ *which is not dominated by any distribution* $\nu$ *with a polynomial-time computable* $\nu'$.

An alternative proof of this theorem can be done using a weaker condition, namely P $\neq$ NP [Yam97].

### 5.1.2 Average case complexity classes

In average case complexity theory, a computational problem is a pair $(L, \mu)$ where $L \subseteq \Sigma^*$ and $\mu$ is a probability distribution. The time and space complexity of an algorithm for that problem is measured under the assumption that the inputs occur according to the given distribution. So one natural notion of complexity classes [Lev86, BDCGL92] is the combination of existing worst case complexity classes $\mathcal{C}$ and sets $\mathcal{F}$ of distributions.

Schuler and Yamakami [SY96] slightly refined and modified the Ben-David *et al.* [BDCGL92] notation $\langle \mathcal{C}, \mathcal{F} \rangle$ and introduced an average case complexity class Dist$\langle \mathcal{C}, \mathcal{F} \rangle$.

**Definition 5.1.19 (Distributional Complexity Classes)** *Let* $\mathcal{C}$ *be a complexity class and* $\mathcal{F}$ *be a class of distributions.* Dist$\langle \mathcal{C}, \mathcal{F} \rangle$ *is the set* $\{(D, \mu) : D \in \mathcal{C}$ *and* $\mu \in \mathcal{F}\}$.

**Note:** to simplify notation we will follow [SY96] convention and let $*$ denote the set of *all* density functions.

Another type of average case complexity classes is defined by forcing the resource bounds of the complexity class to be taken with respect to the given distribution. This classes where introduced by Schuler and Yamakami [SY96], they used a characterization of polynomial on $\mu$-average given by Schapire [Sch90] since it can easily be extended to define the notion of "$t$ on $\mu$-average" for an arbitrary function $t$.

**Definition 5.1.20 ([Sch90])** *Let $t$ be a function on $\mathbf{R}^+$ and let $\mu$ be a distribution. Let $g$ be a function from $\Sigma^*$ to $\mathbf{R}^{+\infty}$. The function $g$ is $t$ on $\mu$-average if $\mu'(\{x : g(x) > t(|x| \times r)\}) < \frac{1}{r}$ for any positive real number $r$.*

Schapire has proved that if we restrict $t$ to be a polynomial, we obtain Levin's notion of polynomial on average.

**Definition 5.1.21 (Domination Relations)** *Let $\mu$ and $\nu$ be any two distributions. We say that $\mu$ average $t$ dominates $\nu$ if there exists a function $t' : \Sigma^* \to \mathbf{R}^+$ such that $t'$ is $t$ on $\mu$-average and $\nu$ $t'$-dominates $\mu$.*

**Definition 5.1.22 (Average Polynomial Domination)** *Let $\mu$ and $\nu$ be two distributions. $\mu$ average polynomially dominates $\nu$ (avp-dominates) if there exists a polynomial $t$ such that $\mu$ average $t$-dominates $\nu$.*

**Definition 5.1.23 (Average Case Complexity Classes)** *Let $t$ be a function on $\mathbf{N}$ and let $\mathcal{F}$ be a class of density functions. Time bounded* **average complexity classes** *are defined as follows:*

1. $\mathrm{Aver}\langle \mathrm{DTIME}(t), \mathcal{F} \rangle = \{(D, \mu) : \mu \in \mathcal{F}$ *and* $D = L(M)$ *for a deterministic Turing machine $M$ which is $t$-time bounded on $\mu$-average*$\}$.
2. $\mathrm{Aver}\langle \mathrm{NTIME}(t), \mathcal{F} \rangle = \{(D, \mu) : \mu \in \mathcal{F}$ *and* $D = L(M)$ *for a nondeterministic Turing machine $M$ which is $t$-time bounded on $\mu$-average*$\}$.
3. $\mathrm{Aver}\langle \mathrm{BPTIME}(t), \mathcal{F} \rangle = \{(D, \mu) : \mu \in \mathcal{F}$ *and* $D = L(M)$ *for a bounded error probabilistic Turing machine $M$ which is $t$-time bounded on $\mu$-average*$\}$.

Using the above definitions, we can consider average case analogues of known time-space bounded complexity classes. Levin [Lev86] implicitly defined the average case

complexity classes Average-P (Aver$\langle$P, P-comp$\rangle$) and Dist-NP (Dist$\langle$NP, P-comp$\rangle$) as analogues of the worst case complexity classes P and NP. An interesting question posed by Levin is whether Average-P contains Dist-NP; this question is still open. However, Schuler and Yamakami [SY96], using results from [BGS75], proved that Dist$\langle$NP, $\mathcal{F}\rangle \subseteq$ Aver$\langle$P, $\mathcal{F}\rangle$ and Dist$\langle$NP, $\mathcal{F}\rangle \not\subseteq$ Aver$\langle$P, $\mathcal{F}\rangle$ in some relativized worlds. So any technique that relativize will not solve this question.

Note however that Dist-NP does not correspond to NP in the same sense that Average-P corresponds to P, since a problem in Dist-NP is not necessarily allowed to be in "average NP", but must be in NP.

We will start by prove some fundamental relations among average complexity classes. By the definition of $t$ on $\mu$-average, Dist$\langle$DTIME, $\mathcal{F}\rangle \subseteq$ Aver$\langle$DTIME, $\mathcal{F}\rangle$. Similar inclusions hold for other average complexity classes.

**Proposition 5.1.24 ([SY96])** *Let* $\mathcal{C} \in \{\mathrm{DTIME}(t), \mathrm{NTIME}(t), \mathrm{BPTIME}(t)\}$ *for some increasing function* $t$ *on* $\mathbf{N}$*, and let* $\mathcal{F}$ *be a set of distributions. Then,* Dist$\langle \mathcal{C}, \mathcal{F}\rangle$ *is included in* Aver$\langle \mathcal{C}, \mathcal{F}\rangle$.

**Proposition 5.1.25 ([SY96])** *Let* $\mathcal{F}$ *be a set of distributions and assume that* $\mu, \nu \in \mathcal{F}$*. Let* $\mathcal{C}$ *be one of the following classes:* L, P, NP, BPP, EXP*. If* $(D, \nu) \in$ Aver$\langle \mathcal{C}, \mathcal{F}\rangle$ *and* $\nu$ *avp-dominates* $\mu$*, then* $(D, \mu) \in$ Aver$\langle \mathcal{C}, \mathcal{F}\rangle$.

A tally set is a subset of $\{0\}^*$ and $\nu_{tally}$ is the standard distribution that is positive only on $\{0\}^*$. A worst case complexity class $\mathcal{C}$ is closed under disjoint union $\oplus$ if, for any sets $A$ and $B$ in $\mathcal{C}$, $A \oplus B$ is in $\mathcal{C}$.

**Proposition 5.1.26 ([SY96])** *Let* $\mathcal{C}$ *and* $\mathcal{D}$ *be two complexity classes, and let* $\mathcal{F}$ *be a set of distributions with* $\nu_{tally} \in \mathcal{F}$*. Assume that* $\mathcal{D}$ *contains the set* $\{0\}^*$ *and is closed under disjoint union. If* $REC - \mathcal{D} \neq \emptyset$ *and* $\mathrm{DTIME}(O(n)) \subseteq \mathcal{C}$*, then* Aver$\langle \mathcal{C}, \mathcal{F}\rangle \not\subseteq$ Dist$\langle \mathcal{C}, *\rangle$.

As a corollary, we get a result by Wang and Belanger [WB95] regarding the separation between Dist$\langle$NP, $*\rangle$ and Aver$\langle$P, $*\rangle$.

**Corollary 5.1.27 ([WB95])** Aver$\langle$P, $\mathcal{F}\rangle \not\subseteq$ Dist$\langle$NP, $*\rangle$ *for any set* $\mathcal{F}$ *of distributions with* $\nu_{tally} \in \mathcal{F}$.

The next theorem proved in [SY96] shows that in the average case setting deterministic computation time is distinct from nondeterministic computation time, i.e., it shows a separation between $\mathrm{Aver}\langle \mathrm{P}, * \rangle$ and $\mathrm{Aver}\langle \mathrm{NP}, * \rangle$.

**Theorem 5.1.28 ([SY96])** $\mathrm{Aver}\langle \mathrm{P}, * \rangle \neq \mathrm{Aver}\langle \mathrm{NP}, * \rangle$.

In average case complexity theory we miss one property, which is used in worst case complexity theory, namely the constructibility of the time-bounds. It is impossible to enumerate all functions which are polynomial on average, since the question whether a function is polynomial on average or not, depends on the choice of the probability distribution. So the well known diagonalization technique cannot be applied directly in average case complexity.

**Example 5.1.29** *In worst case complexity theory if $A$ and $B$ are in* P*, then $A \oplus B$ (disjoint union), is in* P*. In the average case setting, we must consider the distributions as well as the sets of strings.*

### 5.1.3 Reducibility

The worst case reduction of problems is based on two essential requirements, efficiency and validity. In the distributional case it is also required that the reduction "preserve" the probability distribution, i.e., it is necessary to ensure that the reduction $f$ from a distributional problem $(A, \mu)$ to a distributional problem $(B, \nu)$ should not reduce common instances of $A$ to rare instances of $B$, and that the distribution induced on $B$ by $\mu$ should not be "to far" from $\nu$ . This means that the induced weight on the output $y = f(x)$ should be bounded above, within a polynomial factor, by the weight on $y$ according to the distribution of $B$. Namely, $f(\mu)(y) \leq |y|^{O(1)}\nu(y)$. Now if $|y| \leq |x|^{O(1)}$, then there exist a distribution $\mu_1$ such that, for all $x$, it holds that $\mu(x) \leq |x|^{O(1)}\mu_1(x)$ and $\nu(f(x)) = f(\mu_1)(f(x))$, the converse is also true if $x$ is bounded by a polynomial in $|f(x)|$[Gur91]. This conditions are the base for the reductions.

**Definition 5.1.30** *A distributional problem $(A, \mu)$ is* polynomial-time *reducible to a distributional problem $(B, \nu)$, $(A, \mu) \propto (B, \nu)$, if there exists a function $f : \Sigma^* \to \Sigma^*$ such that:*

1. Efficiency: *$f$ is computable in polynomial-time.*

2. Validity: *$x \in A$ if and only if $f(x) \in B$.*

3. Domination: *There exists a constant $c > 0$ such that,*

$$\nu'(y) \geq \frac{1}{|y|^c} \times \sum_{x \in f^{-1}(y)} \mu'(x)$$

The last condition states that $\nu$ dominates the induced distribution $\mu_f$ defined by

$$\mu'_f(y) = \sum_{x \in f^{-1}(y)} \mu'(x).$$

We can weaken the notion of reducibility by requiring the function $f$ to be *average polynomial* time computable instead of *polynomial* time computable.

**Definition 5.1.31** *A distributional problem $(A, \mu)$ is* average polynomial-time *reducible to a distributional problem $(B, \nu)$, $(A, \mu) \propto (B, \nu)$, if there exists a function $f : \Sigma^* \to \Sigma^*$ such that:*

1. Efficiency: *with respect to $\mu$ $f$ can be computed in average polynomial-time.*

2. Validity: *$x \in A$ if and only if $f(x) \in B$.*

3. Domination: *There exists a constant $c > 0$ such that,*

$$\nu'(y) \geq \frac{1}{|y|^c} \times \sum_{x \in f^{-1}(y)} \mu'(x)$$

Average polynomial time reductions can be used to establish completeness results for the class of distributional problems that are solvable by nondeterministic algorithms in average polynomial time. It is not know if the same holds using polynomial-time reductions, which lead us to suspect that average polynomial-time reductions are more powerful than polynomial-time reductions.

### 5.1.4 Average Case Completeness

Using average polynomial-time reducibility, Levin was able to show that there is a problem (tiling) which is complete for the class Dist⟨NP, P-comp⟩, i.e., a problem to which every other problem in Dist⟨NP, P-comp⟩ is reducible. Thus, he has succeeded in identifying a "hardest" problem in Dist⟨NP, P-comp⟩ which can only be polynomial on average if every other problem in the class is as well.

Proving completeness results for distributional problems is not as easy as for the worst case complexity classes. The difficulty is that we have to reduce all distributional problems with *different distributions* to one single distributional problem with a *specific distribution.* The usual (worst case) reduction is $x \rightarrow (M_D, x, 1^{P_D(|x|)})$, where $D$ is the NP problem we want to reduce and $M_D$ is the non-deterministic machine that solves $D$ in time $P_D(n)$ on input length $n$. This reduction is valid for every distributional problem, but for some distributions it fails the domination condition: an extremely common instance with probability $|x|^{-2}$, is mapped to a far rarer instance with probability proportional to $|x|^{-2}2^{-|x|}$.

Levin [Lev86] showed that a tilling problem under a near uniform distribution is complete for Dist⟨NP, P-comp⟩; a more detailed proof can be found in [Gur91]. Levin also implicitly showed that a distributional halting problem is complete for Dist⟨NP, P-comp⟩. Gurevich [Gur91] explicitly defined such a problem and provided a simple proof of its completeness.

**Definition 5.1.32 (Distributional Tiling Problem (DT))**

- Instance: *A tile is a square with a symbol on each side that may not be rotated or turned over. By a tilling of an $n \times n$ square we mean an arrangement of $n^2$ tiles covering the square in which the symbols on the common sides of adjacent tiles are the same. The Distributional Tiling Problem consists of binary strings $T$, $1^n$, and $S$ as instances, where $T$ is a finite set of tiles, $n$ is an integer, and $S = s_1 s_2 s_3 ... s_k (k \leq n)$ is a sequence of tiles that match each other, i.e., the symbol on the right side of $s_i$ is the same as that on the left side of $s_{i+1}$.*
- Question: *Can $S$ be extended to tile an $n \times n$ square using tiles from $T$.*
- Distribution: *Denote by $BT$ the set of all positive instances $(T, 1^n, S)$. The prob-*

*ability distribution $\mu_{BT}$ on instance $(T, 1^n, S)$ is proportional to $Pr[T]n^{-2}Pr[S]$, where $Pr[T]$ is the probability of $T$ and $Pr[S]$ is the probability of choosing $S$, where $S$ is chosen by choosing $k$ at random with probability $\frac{1}{n}$, choosing the first tile $s_1$ at random from $T$, and choosing the $s_i$ sequentially and uniformly at random from those tiles in $T$ that matches $s_{i-1}$.*

**Theorem 5.1.33 ([Lev86])** *The* **DT** *problem is complete for* Dist⟨NP, P-comp⟩.

**Definition 5.1.34 (Bounded Halting Problem (BH))**

- Instance: *An encoding $M$ of a nondeterministic Turing machine, a string $x$, and a number $t$ in unary notation.*
- Question: *Does the machine encoded by $M$ accept $x$ within $t$ steps?*
- Distribution: *The values of $t$, $|M|$ and $|x|$ are chosen first with probability proportional to an inverse quadratic. Then, $M$ and $x$ are chosen uniformly from all strings of the given length. Thus, $\mu'(x) = |M|^{-2}2^{-|M|}|x|^{-2}2^{-|x|}t^{-2}$.*

**Theorem 5.1.35 ([Gur91])** *The* **BH** *problem is complete for* Dist⟨NP, P-comp⟩.

**Proof.** (*Sketch*) That BH is in Dist⟨NP, P-comp⟩ is easily verified. Let $(D, \nu)$ be any distributional problem in Dist⟨NP, P-comp⟩, we wish to reduce $(D, \nu)$ to $(BH, \mu)$. The usual worst case reduction fails the domination condition. The essence of the problem is that $\mu'$ gives low probability to long strings, whereas an arbitrary distribution can give them high probability. To overcome this problem we use the encoding given by the Distribution Controlling Lemma 5.1.13 and map with high probability long strings to short strings, which get high probability in $\mu'$. This encoding $C_\mu$ maps a string $x$ into a code of length bounded above by $\log_2 \frac{1}{\mu'(x)}$. Now consider the following reduction of $(D, \nu)$ to $(BH, \mu)$:

$$x \to (M_{D,\nu}, C_\nu(x), 1^{P_{D,\nu}(|x|)})$$

where $M_{D,\nu}$ is a non-deterministic machine that on input $y$ non-deterministically guesses $x$ such that $C_\nu(x) = y$, and then runs $M_D$ on $x$. The polynomial $P_{D,\nu}$ is defined as $P_D(n) + P_C(n)$, where $P_D(n)$ is a polynomial bounding the running time of $M_D$ on acceptable inputs of length $n$ and $P_C(n)$ is a polynomial bounding the running

time of the encoding algorithm. It remains to show that this reduction satisfies the following requirements:

1. *Efficiency:* The description of $M_{D,\nu}$ has fixed length and by the Distribution Controlling Lemma 5.1.13 $C_\nu$ is polynomial-time computable.

2. *Validity:* By construction we have that $D(x) = 1$ if and only if there exists a computation of $M_{D,\nu}$ that on input $C_\nu$ halts with output 1 within $P_{d,\nu}(|x|)$ steps.

3. *Domination:* It suffices to consider instances of BH which have a pre-image under the reduction. Since the coding is one-to-one, each such image has a unique pre-image. By the definition of $\mu$,

$$\mu'(M_{D,\nu}, C_\nu(x), 1^{P_{D,\nu}(|x|)}) = c\frac{1}{P_{D,\nu}(|x|)^2}\frac{1}{|C_\nu(x)|^2 2^{|C_\nu(x)|}}$$

where $c = \frac{1}{|M_{D,\nu}|^2}2^{|M_{D,\nu}|}$ is a constant independent of $x$. By the Distribution Controlling Lemma we know that

$$\nu'(x) \leq 2 \times 2^{-|C_\nu(x)|}.$$

Hence,

$$\mu'(M_{D,\nu}, C_\nu(x), 1^{P_{D,\nu}(|x|)}) \geq c\frac{1}{P_{D,\nu}(|x|)^2}\frac{1}{|C_\nu(x)|^2}\frac{\nu'(x)}{2} > \frac{2}{2P_{D,\nu}(|x|)^2|C_\nu(x)|^2}\nu'(x).$$

◇

**Theorem 5.1.36** *Every problem complete for* Dist⟨NP, P-comp⟩ *is also complete for* Dist⟨NP, P-samp⟩

## 5.2 Average Case complexity under the Universal Distribution

Li and Vitányi [LV97a] proved that one particular distribution, the *universal distribution* or *Solomonoff-Levin distribution*, has the property that, when the inputs to any algorithm are distributed according to this distribution, the average case complexity of the algorithm is of the same order of magnitude as its worst case complexity. The

main result in this chapter, theorem 5.3.2, generalizes this result. Let us begin by studying in some detail the *universal distribution* and Li and Vitányi proof.

We have seen in Definition 5.1.6 that a distribution is *universal for a set of distributions*, if it is in the set and it dominates every distribution in the set.

**Theorem 5.2.1** *The class of recursive semi-distributions has no universal element.*

**Proof.** Suppose that such an universal element $\mu$ exists. Since for each $x \in \Sigma^*$ there is a recursive semi-distribution $\mu_x$ such that $\mu_x(x) > 0$, then by the universatility of $\mu$ we have $\mu(x) > 0$. As $\mu$ is a semi-distribution then $\sum_x \mu(x) \leq 1$ and the function $\mu$ converges to 0. We can compute a sequence $x_0 < x_1 < ...$ such that

$$\sum_{x_i<x<x_{i+1}} \mu(x) > x_i\mu(x_i),$$

for all $i$. So the function $\nu(x) = x\mu(x)$ for $x = x_1, x_2, ...$ and 0 otherwise, is recursive and also a semi-distribution as $\sum_x \nu(x) \leq 1$. However for each constant $c$ there is an $x$ such that $\nu(x) > c\mu(x)$, which contradicts the universality of $\mu$. ◇

**Definition 5.2.2** *A distribution $\mu$ is called* enumerable, *if the set of points*

$$\{(x, y) : x \in \Sigma^*, y \in Q, \mu(x) > y\}$$

*is recursively enumerable, i.e., $\mu(x)$ can be approximated from below.*

Note that the enumerable distributions include the recursive ones. Zvonkin and Levin have shown [ZL70a] that we can effectively enumerate all enumerable distributions, $\mu_1, \mu_2, ....$ In particular Levin has shown the following result.

**Theorem 5.2.3** *There is a universal enumerable discrete semi-distribution. We denote it by* **m**.

**Proof.** (*Sketch*) The theorem is proved in two steps:

1. Prove that the enumerable discrete semi-distributions can be effectively enumerated as

$$\mu_1, \mu_2, ...$$

This is done by first enumerating all enumerable functions and then effectively changing the enumerable functions to enumerable discrete semi-distributions, leaving the functions that were already discrete semi-distributions unchanged.

2. Show that
$$\nu(x) = \sum_{n \geq 1} \alpha(n)\mu_n(x),$$
with $\sum \alpha(n) \leq 1$ where $\alpha(n) > 0$, for all $n$ is universal.

◇

In the definition of $\mathbf{m}(x)$ as $\sum_n \alpha(n)\mu_n(x)$ we must chose an $\alpha$ such that the resulting $\mathbf{m}$ is enumerable. In particular, we can define
$$\mathbf{m}(x) = \sum_{n \geq 1} 2^{-K(n)}\mu_n(x).$$
This choice yields the domination relation $\mathbf{m}(x) \geq 2^{-K(n)}\mu_n(x)$. One important property is that under $\mathbf{m}$, easily describable objects have high probability, and complex or random objects have low probability. It embodies Occam's Razor principle, which says we should prefer simple explanations over complicated ones.

Solomonoff, trying to predict continuations of each finite prefix of infinite binary sequences [Sol64], introduced the universal distribution. We can view the discrete probability density $\mathbf{m}$ as the *a priori* probability of finite objects in absence of any knowledge about them.

**Lemma 5.2.4** *The function* $\mathbf{m}$ *is not recursive and* $\sum_x \mathbf{m}(x) < 1$.

**Proof.** Suppose that $\mathbf{m}$ is recursive. By Theorem 5.2.3 it should be universal for the class of recursive semi-distributions that contradicts Theorem 5.2.1.

An enumerable semi-distribution is also a probability distribution, so it is recursive. Since $\mathbf{m}$ is enumerable but not recursive this implies $\sum_x \mathbf{m}(x) < 1$. ◇

**Definition 5.2.5** *The universal a priori probability on the positive integers is defined as*
$$Q_U(x) = \sum_{U(p)=x} 2^{-|p|}$$
*where $U$ is the reference prefix machine.*

In other words, $Q_U(x)$ is the probability that $U$ computes $x$ if its input is a sequence of tosses of a fair coin.

**Definition 5.2.6** *The algorithmic probability $R(x)$ of $x$ is*

$$R(x) = 2^{-K(x)}$$

Now we will state a very strong result, proved by Levin, relating three quite different formalizations of concepts, $\mathbf{m}(x)$, $Q(x)$ and $R(x)$: all coincide up to an independent fixed multiplicative constant. In [LV97b] a detailed proof can be found.

**Theorem 5.2.7** *There is a constant $c$ such that for every $x$,*

$$-\log \mathbf{m}(x) = -\log Q_U(x) = K(x)$$

*with equality up to an additive constant $c$.*

In [LV97a] Li and Vitányi proved that, under the universal distribution, the average case complexity of any machine is at most a constant factor (depending only on the particular machine) smaller than its worst case complexity. More precisely, for some constant $c$,

$$\sum_{x\in\Sigma^n} \frac{\mathbf{m}(x)}{\sum_{y\in\Sigma^n} \mathbf{m}(y)} T_A(x) \geq c \max_{x\in\Sigma^n} T_A(x).$$

We will present the proof given in [Mil91] which is equivalent to the original proof in [LV97a]. But the different presentation will help us to analyze the result in more depth. We need some notation. Given an algorithm $A$, we define $T_A(x)$ for $x \in \Sigma^*$ to be its running time on the binary string $x$. $T_A^\omega(n) = \max_{|x|=n} T_A(x)$ is the algorithm worst case running time. $\omega_A(n)$ is the lexicographic least string in $\Sigma^n$ with $T(\omega_A(n)) = T_A^\omega(n)$ and $T_A^a(\mu, n) = \sum_{|x|=n} \mu_n(x) T_A(x)$ is the algorithm's average case running time with respect to the distribution $\mu$.

**Theorem 5.2.8 ([LV97a])** *Let $A$ be any algorithm, that halts for all inputs in $\mathbf{N}$. Let the inputs to $A$ be distributed according to $\mathbf{m}$. Then the average case time complexity is of the same order of magnitude as the corresponding worst-case time complexity.*

**Proof.** Consider the Turing machine $M$ with the following behavior:

*Read the prefix* $1^i0$, $i \geq 0$ *from the tape.*
*Simulate the universal machine* $U$ *on the rest of the tape.*
`if` $U$ *halts* `then`

$n = |U$*'s output*$|$

*simulate* $A_i$ *on all inputs of length* $n$, *finding* [1] *the*

*lexicographically least worst case output,* $\omega_{A_i}(n)$.

`output` $\omega_{A_i}(n)$.
`fi`

Assume $M = M_k$ in an effective enumeration of Turing machines $M_1, M_2, \ldots$ ($U$ is universal for this class), and assume that $i$ is the index of an algorithm, i.e., that $A_i$ halts on all inputs. If $U$ is started with the tape $1^k01^i0t$, where $U$, started on $t$, outputs a string of length $n$, $\omega_{A_i}(n)$. The events of reading $1^k01^i0t$ and $t$ are independent, this means that for all $n$

$$\mathbf{m}(\omega_{A_i}(n)) \geq 2^{-k-i-2}\mathbf{m}(\Sigma^n).$$

But then

$$T^a_{A_i}(\mathbf{m}) \geq \frac{\mathbf{m}(\omega_{A_i}(n))}{\mathbf{m}(\Sigma^n)} T^\omega_{A_i}(n) \tag{5.1}$$

$$\geq 2^{-k-i-2} T^\omega_{A_i}(n). \tag{5.2}$$

◇

As noted by Miltersen [Mil91] this result suggests that generally inputs which require a lot of time contain lots of regularities (pattern). By examining the proof of Theorem 5.2.8 we see that in fact the worst case input has a lot of pattern. In the proof, the worst case input, of length $n$, to $A_i$ can be described in the following way:

*"The worst case input, of length* $n$, *to* $A_i$*."*

Thus $\omega_{A_i}(n)$ has a short description, i.e., worst case inputs have a pattern, but it is the very pattern of being a worst case input. The problem with the pattern is that it takes

[1] This may not halt.

an exponential time to get the result from the description. Thus we can summarize that the regularities is difficult, i.e., it is deep.

Miltersen [Mil91] has extended the result of Li and Vitányi [LV97a] to subclasses of algorithms $\mathcal{A}$ with a fixed upper time bound. He calls a distribution *malign* for $\mathcal{A}$ if the average complexity of any algorithm in $\mathcal{A}$ is of the same order of magnitude as its worst case complexity, and has proved the existence of such a distribution. This distribution is computable in exponential time and malign for the class P. Furthermore, no distribution computable in polynomial-time has this property if the probability of inputs strings do not decrease too fast to 0 with respect to their length. If we replace computability by samplability it can be showed that NP has malign distributions that can be generated, sampled, in polynomial-time [BDCGL92].

### Time limited Universal Distributions

The main drawback of $\mathbf{m}$ is that it is not computable. So we can try to scale this theory to a more feasible domain. We will study the *polynomial-time bounded approximation* of $\mathbf{m}$ and show that it has similar domination properties with respect to the polynomial time distributions. For each time bound $t(n)$, a function $t'(n) = nt(n)$ is constructed such that $\mathbf{m}^{t'}$ is computable in time $nt(n)2^n$ and multiplicatively dominates each probability distribution $\mu$ with a distribution function $\mu^*$ computable in time $t(n)$.

**Definition 5.2.9** *The t-time bounded universal distribution,* $\mathbf{m}^t$ *is given by*

$$\begin{aligned} \mathbf{m}^t(x) &= 2^{-K^t(x)} \\ \mathbf{m}^{*t}(x) &= \sum_{y \leq x} \mathbf{m}^t(x). \end{aligned}$$

Note that $\mathbf{m}^t$ approaches $\mathbf{m}$ as $t \to \infty$. When there is no confusion, we call this time bounded version also the universal distribution. The universality of $\mathbf{m}^t$ is justified by the property that it very closely dominates distributions which can be computed in certain time bound [LV97b]. However we do not know if $\mathbf{m}^t$ itself can be computed in that amount of time.

**Definition 5.2.10** *Let $t$ be a time constructible function. A probability distribution function $\mu$ on $\{0,1\}^*$ is said to be t-time computable, if there is a deterministic Turing*

*machine that on every input $x$ and a positive integer $k$, runs in time $t(|x|+k)$, and outputs a fraction $y$ such that $|\mu^*(x) - y| \leq 2^{-k}$.*

**Theorem 5.2.11 ([LV97b])** $\mathbf{m}^t$ *dominates any $t/n$-time computable distribution.*

**Proof.** (*Sketch*) Let $\mu$ be a $t/n$-time computable distribution and let $\mu^*$ denotes the distribution of $\mu$. We will show that for any $x \in \Sigma^n$, $K^t(x) \leq -\log(\mu(x)) + C_\mu$ for a constant $C_\mu$ which depends on $\mu$. Let $B_i = \{x \in \Sigma^n | 2^{-(i+1)} \leq \mu(x) < 2^{-i}\}$. Since for any $x$ in $B_i$, $\mu(x) \geq 2^{-i}$, we have that $|B_i| \leq 2^i$. Consider the real interval $[0,1]$. Divide it into intervals of size $2^{-i}$. Since $\mu(x) \geq 2^{-i}$, we have that for any $j, 0 \leq j \leq 2^i$, the $j^{th}$ interval $[j2^{-i}, (j+1)2^{-i}]$ will have at most one $x \in B_i$ such that $\mu(x) \in [j2^{-i}, (j+1)2^{-i}]$. Since $\mu$ is $t/n$-computable, for any $x \in B_i$, given $j$, we can do a binary search to output the unique $x$ satisfying $\mu(x) \in [j2^{-i}, (j+1)2^{-i}]$. This involves computing $\mu^*$ correct up to $2^{-(i+1)}$. So the total running time of the process will be bounded by $O((t/n)n)$. ◇

Note that in the proof $\mathbf{m}^t$ very strongly dominates $t/n$-time computable distributions, in the sense that $\mathbf{m}^t(x) \geq \frac{1}{2^{C_\mu}}\mu(x)$.

### Dominating Time Bounded Universal distributions

It is then natural to ask if there exists a polynomial-time computable distribution dominating $\mathbf{m}^t$.

Based on the connection between pseudo-random generators and time-bounded Kolmogorov complexity, Schuler [Sch99] showed that if such a $P$-computable distribution dominating $\mathbf{m}^t$ exists then no polynomially secure pseudo-random generators exists. Pseudo-random generators are efficiently computable functions which stretches a seed into a long string so that for a random input the output looks random for a resource-bounded machine.

**Definition 5.2.12** *A deterministic polynomial time algorithm $G$ is called a pseudo-random generator if there exists a stretching function $l : \mathbf{N} \to \mathbf{N}$, so that letting $U_m$ denote the uniform distribution over $\{0,1\}^m$, for any probabilistic polynomial time algorithm $A$, for any positive polynomial $p$, and for all sufficiently large $n$'s*

$$|Pr_{s \in U_n}[A(G(s)) = 1] - Pr_{r \in U_{l(n)}}[A(r) = 1]| < \frac{1}{p(n)}$$

So by definition of pseudo-random generator we can not generate strings of high Kolmogorov complexity (unbounded or polynomial time bounded) from short random strings. But if there is a distribution $\mu$ dominating $\mathbf{m}^t$, then we have $K^t(x) > -log(\mu(x))$ up to some additive factors.

If $\mu$ is polynomial-time computable, this conflict gives an algorithm for fooling the pseudo-random generator. We give the formal argument next.

**Theorem 5.2.13 ([Sch99])** *If there exists a polynomial time computable distribution that dominates $\mathbf{m}^t$ then pseudo-random generators do not exist.*

**Proof.** Let $\mu$ be a polynomial-time computable distribution that dominates $\mathbf{m}^t$. Thus there is a constant $d$ such that $\mu(y) \geq \frac{1}{|y|^d}\mathbf{m}^t(y)$. From the definition of $\mathbf{m}^t$, we have $K^t(y) \geq -log(\mu(y)) - d\log|y|$.

**Claim 5.2.14** *Let $G$ be a pseudo-random generator with stretch function $l(n) = cn$, $c > 1$, which is computable in time $n^k$. Let $R(G)$ denote the range of $G$. Then for any $t > n^{k+1}$ we have for any $y \in R(G)$, $K^t(y) \leq n + O(1)$, note that $|y| = cn$ with $c > 1$.*

Now we can give a polynomial time procedure which accepts all strings in $R(G)$, namely compute $-\log(\mu(y))$ and accept if and only if $-\log(\mu(y)) \leq n + c_1$ where $c_1 = -d\log|y|$. Since only a small fraction of strings will have weight $\geq 2^{-n-c_1}$, the acceptance probability is very small. On the other hand all the strings in $R(G)$ are accepted. Suppose that some $y \in R(G)$ is rejected, then $-\log(\mu(y)) > n + c_1$. This means $K^t(y) \geq -log(\mu(y)) - d\log|y| > n$, a contradiction. ◇

It is known that pseudo-random generators exists if and only if certain one-way functions exists [HILL99]. (Easily computable functions that are hard to invert over a significant fraction of their range). Thus existence of these type of one-way functions implies that there are no polynomial-time computable distributions that dominates $\mathbf{m}^t$.

In the next section we show that there exists a samplable distribution dominating $\mathbf{m}^t$ (Lemma 5.3.5).

## 5.3 Computational Depth and Average polynomial-time

Since the universal distribution is not recursive, Theorem 5.2.8 fits poorly with the traditional average case complexity analysis, where the distribution must be "simple" (usually considering polynomial-time computable or samplable distributions). Therefore, unless we can derive some kind of time bound version of Theorem 5.2.8, the two subjects seem unrelated.

We state our main theorem which relates computational depth to average polynomial-time, using the time bounded universal distribution.

**Definition 5.3.1** *Let $x \in \Sigma^*$. The t-time bounded computational depth of $x$ is*

$$depth_t(x) = K^t(x) - K(x).$$

**Theorem 5.3.2** *Let $T$ be a constructible time bound. Then for any time constructible $t$, the following statements are equivalent.*

1. $T(x) \in 2^{O(depth_t(x))} \cdot \text{poly}(|x|)$
2. *$T$ is polynomial on $\mathbf{m}^t$-average.*

Before giving the proof of this result, let us give some interpretations of it.

Li and Vitányi [LV97a] showed that the average case complexity of algorithms with respect to the (unbounded) universal distribution is the same as its worst case complexity. Rephrasing this connection in the setting of average polynomial-time we can make the following statement.

**Theorem 5.3.3 (Li-Vitányi)** *Let $T$ be a constructible time bound. The following statements are equivalent*

1. *$T(x)$ is bounded by a polynomial in $|x|$.*

*2. $T$ is polynomial on* $\mathbf{m}$*-average.*

As $t \to \infty$, $K^t$ approaches $K$. So $depth_t$ approaches 0 and $\mathbf{m^t}$ approaches $\mathbf{m}$. Hence our main theorem can be seen as a generalization of Li and Vitányi's theorem.

We can apply the the implication $(1 \Rightarrow 2)$ of the main theorem in the following way. Let $M$ be a Turing machine and let $L(M)$ denote the language accepted by $M$. Let $T_M$ denotes its running time. So if $T_M(x) \in 2^{O(depth_t(x))} \cdot \text{poly}(|x|)$ then $(L(M), \mu)$ is in Average-P for any $\mu$ which is computable in time $t/n$. The following corollary follows from our main Theorem 5.3.2 and the universality of $\mathbf{m}^t$ (Theorem 5.2.11).

**Corollary 5.3.4** *Let $M$ be a deterministic Turing machine whose running time is bounded by $2^{O(depth_t(x))} \cdot \text{poly}(|x|)$. Then for any $t/n$-computable distribution $\mu$, the pair $(L(M), \mu)$ is in Average-*P.

Hence a sufficient condition for a language $L$ (accepted by $M$) to be in Average-P with respect to all polynomial-time computable distributions is that the running time of $M$ is exponentially bounded in $depth_t$ for all polynomials $t$. An obvious question that arises is whether this condition is necessary. We partially answer this question by exhibiting an efficiently *samplable* distribution $\mu_t$ that dominates $\mathbf{m}^t$. Hence if $(L(M), \mu_t)$ is in Average-P then $(L(M), \mathbf{m}^t)$ is also in Average-P. From the implication $(2 \Rightarrow 1)$ of the main theorem, we have that $T_M(x) \in 2^{O(depth_t(x))} \cdot \text{poly}(|x|)$.

**Lemma 5.3.5** *For any polynomial $t$, there is a* P*-samplable distribution $\mu$ which dominates $\mathbf{m}^t$.*

**Proof.** (*Sketch*) We will define a samplable distribution $\mu_t$ by prescribing a sampling algorithm for $\mu_t$ as follows. Let $U$ be the universal machine.

Sample $n \in \mathbf{N}$ with probability $\frac{1}{n^2}$
Sample $1 \leq j \leq n$ with probability $1/n$
Sample uniformly $y \in \Sigma^j$
Run $U(y)$ for $t$ steps and output if $U$ stops and outputs a string in $\Sigma^n$

For any string $x$ of length $n$, $K^t(x) \leq n$. Hence it is clear that the probability that $x$ is at least $\frac{1}{n^3}2^{-K^t(x)}$. ◇

From Lemma 5.3.5, we get that if a machine runs in time polynomial on average for all P-samplable distributions then it runs in time exponential in its depth.

**Corollary 5.3.6** *Let $M$ be a machine which runs in time $T_M$. Suppose for all distributions $\mu$ in* P*-samplable, $T$ is polynomial on $\mu$-average, then $T_M(x) \in 2^{O(depth_t(x))} \cdot \text{poly}(|x|)$.*

We now prove our main Theorem.

**Proof.** (*Theorem 5.3.2*) (1 $\Rightarrow$ 2). We will show that the statement 1 implies that $T(x)$ is polynomial on $\mathbf{m}^t$-average. Let $T(x) \in 2^{O(depth_t(x))}.\text{poly}(|x|)$. Because of the closure properties of functions which are polynomial on average, it is enough to show that the function $T'(x) = 2^{depth_t(x)}$ is polynomial on $\mathbf{m}^t$-average. This essentially follows from the definitions and Kraft's inequality. As we now explain in more detail. Consider the sum

$$\begin{aligned}
\sum_{x\in\Sigma^*} \frac{T'(x)}{|x|}\mathbf{m}^t(x) &= \sum_{x\in\Sigma^*} \frac{2^{depth_t(x)}}{|x|}2^{-K^t(x)} \\
&= \sum_{x\in\Sigma^*} \frac{2^{K^t(x)-K(x)}}{|x|}2^{-K^t(x)} \\
&\leq \sum_{x\in\Sigma^*} \frac{2^{-K(x)}}{|x|} < \sum_{x\in\Sigma^*} 2^{-K(x)} < 1
\end{aligned}$$

The last inequality is the Kraft's inequality.

(2 $\Rightarrow$ 1) Let $T(x)$ be a time constructible function which is polynomial on $\mathbf{m}^t$-average. Then for some $\epsilon > 0$ we have

$$\sum_{x\in\Sigma^*} \frac{T(x)^\epsilon}{|x|}\mathbf{m}^t(x) < 1$$

Define $S_{i,j,n} = \{x \in \Sigma^n | 2^i \leq T(x) < 2^{i+1} \text{ and } K^t(x) = j\}$. Let $2^r$ be the approximate size of $S_{i,j,n}$. Then the Kolmogorov complexity of elements in $S_{i,j,n}$ is $r$ up to an additive $\log n$ factor. The following claim states this fact more formally.

**Claim 5.3.7** *For $i, j \leq n^2$, let $2^r \leq |S_{i,j,n}| < 2^{r+1}$. Then for any $x \in S_{i,j,n}$, $K(x) \leq r + O(\log n)$.*

Consider the above sum restricted to elements in $S_{i,j,n}$.

$$S = \sum_{x \in S_{i,j,n}} \frac{T(x)^\epsilon}{|x|} \mu(x) < 1$$

So we have that $T(x) \geq 2^i$, $\mathbf{m}^t(x) = 2^{-j}$ and there are at least $2^r$ elements in the above sum. Thus $\frac{2^r 2^{i\epsilon} 2^{-j}}{|x|^c}$ is a lower bound of $S$ for some constant $c$. Then

$$\begin{aligned} 1 &> \sum_{x \in S_{i,j,n}} \frac{T(x)^\epsilon}{|x|} \mathbf{m}^t(x) \\ &\geq \frac{2^r . 2^{i\epsilon} . 2^{-j}}{|x|^c} = 2^{i\epsilon + r - j - c \log n} \end{aligned}$$

That is, $i\epsilon + r - j - c \log n < 1$. From Claim 5.3.7, it follows that there is a constant $d$, such that for all $x \in S_{i,j,n}$, $i\epsilon \leq \mathit{depth}_t(x) + d \log |x|$. Hence $T(x) \leq 2^{i+1} \leq 2^{\frac{d}{\epsilon}(\mathit{depth}_t(x) + \log |x|)}$. ◇

# 6

# Conclusions and Further Research

In this thesis we introduced the notion of computational depth in order to measure the amount of non-random or *Useful Information* in a string. We do not believe that there is a single best type of computational depth. Different notions have different properties and applications.

While complexity theory studies the difficulty of solving all instances of a given problem with a single program, Kolmogorov complexity is a more local measure of complexity. It is interesting that such a measure of complexity is a valuable tool in the study of complexity theory [Lap97]. The interplay between the two areas of research occurs on different levels. Kolmogorov complexity can be used as a proof technique to solve problems in complexity theory, or, as we have done in this work, it can be viewed as an independent approach to complexity.
We introduced *shallow sets*, sets containing little nonrandom information, as a generalization of sparse and random sets based on low depth properties of their characteristic sequences. Despite the fact that most sets are shallow, we proved that these sets have very limited computational power and that computable sets reducible to shallow sets must have small circuits. In particular, if NP-complete sets reduce to shallow sets, then the polynomial-time hierarchy collapses.

We have also introduced the notion of *distinguishing computational depth* as a measure of the difficulty between recognizing a string and producing it. We proved that if all the satisfying assignments of a formula $\phi$ have low distinguishing computational depth relative to $\phi$, then we can find an assignment in probabilistic quasi-polynomial

time. An obvious open problem suggested by this result is to study its validity using polynomial time.

**Conjecture 6.1** *Let $\phi$ be a Boolean formulae. If for all assignments $x$ satisfying $\phi$, $D^{p_1,p_2}(x|\phi) \leq O(\log|\phi|)$, where $p_1, p_2$ are polynomials in $|x|$, then there is a probabilistic polynomial algorithm that finds some satisfying assignment.*

One obvious way to prove this conjecture is to try to improve the result from [FL98] which was used in the proof of our result, more specifically we should improve the $O(1)$ in 2.3.32. We emphasize however that the explicit construction of optimal extractors would help the prove of this conjecture.

Li and Vitányi showed that, when the inputs to any algorithm are distributed according to the universal distribution, then the average case complexity of the algorithm is of the same order of magnitude as its worst case complexity. As the universal distribution is not recursive, we proved a time-bounded version of Li and Vitányi's result. We consider another instantiation of computational depth, defined as the difference between $K^t(x)$ and $K(x)$ and proved the following result: the running time $t$ of a machine is polynomial on average with respect to the time-bounded universal distribution if and only if $t$ is bounded by $2^{depth}$. As an interesting corollary we get that, if the running time $t$ of a machine is exponentially bounded in the depth, then $t$ is polynomial on average with respect to all computable distributions of a certain time bound. This result illustrate the strong relation between Kolmogorov complexity, in the form of computational depth, and complexity theory. So we believe that the instantiations studied in this thesis and other instantiations of the concept of computational depth could be useful solving other complexity theory problems.

In this work we have taken a fresh look at sophistication. Koppel claimed that depth and sophistication for all infinite strings are equivalent. However this result is wrong and uses a different definition of depth imposing totality in the functions defining depth. In this thesis we proved that it is in fact possible to have some discrepancy between depth and sophistication. Nevertheless, we consider that sophistication is a significant concept and studied its properties for finite strings. In particular, we proved that there are strings $x$ of length $n$ with sophistication close to $n$.

There can be several ways (model classes) in which regularity is expressed. Kolmogorov has proposed the model class of finite sets, generalized later by Gács, Tromp and

Vitányi to computable probability mass functions. In Chapter 4 we claimed that the most general way to proceed is perhaps to express the regularity as a recursive function. However we now believe that one can go a little further, and that the next step would be to express regularity as a primitive recursive function. Note that the class of primitive recursive functions is a subclass of the recursive functions having an inductive definition. So we can define sophistication based on this functions and use the fact that we can inductively define the class. This seems to be a natural step for further research in this area.

Ideally, in the definition of the sophistication

$$soph_c(x) = \min\{|p| : \text{p is total, } \exists d[U(p,d) = x \text{ and } |p| + |d| \leq K(x) + c]\}$$

we would like to have $|p| + |d| = K(x)$. But, since this is too restrictive, we must allow a little freedom for coding so we really have $|p| + |d| \leq K(x) + c$. But what constant $c$ should we choose? Could a small variation in $c$ dramatically change the sophistication, i.e., is sophistication a robust measure?

**Definition 6.2** *Sophistication is stable if for every fixed universal Turing machine $U$, there is a constant $c_1$ such that for every constant $c_2 > c_1$, there is a constant $c_3$ such that for all $x \in \Sigma^*$*

$$|soph_{c_2}(x) - soph_{c_1}(x)| < c_3.$$

We conjecture that there are strings for which a small change in $c$ could dramatically change the sophistication. This conjecture is based on intuition. Of course, one can have a different intuition. The following example was given by Paul Vitányi in order to justify the stability of sophistication:

**Example 6.3** *For plain complexity we can show that there are $x$'s with complexity $|x| + 1$ (in fact there are a lot such $x$'s). But we can also show that every $x$ has complexity at most $x+c$ for some fixed constant $c$ depending on the reference universal machine. By choice of machine we can set that constant to* every *integer $\geq 1$. Also to* 1, *and also to* 1000000. *So all theorems need to take that unspecified constant into account. Setting it to* 1 *is possible but for a very special reference machine, which will result in the fact that other constants in other arguments get larger again.*
*Thus, one wants to derive universal statements that hold up to an unspecified additive constant. This can also be done for the sophistication.*
*In any case, the Kolmogorov minimal sufficient statistic do not change as long as one*

*takes the additive constant large enough. Of course, if we insist on* $K(x) = |p| + \log|S|$ *or* $K(x) > |p| + \log|S|$ *then the Kolmogorov minimal sufficient statistic may not exist at all and there are unclear regions like* $K(x) = |p| + \log|S| + 50$ *where* $\{x\}$ *may be an implicitly described Kolmogorov minimal sufficient statistic but the set* $\{y : K(y) \leq K(x)\}$ *is not an implicit Kolmogorov minimal sufficient statistic for* $x$.
*Those intermediate regions however, are sensitive to the* precise reference universal machine. *This one wants to avoid in arguments about Kolmogorov complexity. The theorems should always hold independent of the precise reference universal machine chosen, and this* results *in the unspecified additive constant.*

This example shows that we can not fix the constant, in fact it should be large enough to accommodate the finite programs involved. The reason is that the universal Turing machine needs to simulate programs and for each program we need a different constant. Thus, this constant is the description of the Turing machine that the universal Turing machine is going to simulate. So, as this description can be of any size, if we fix it we eliminate some programs. However, the question remains: for an arbitrary constant $c$, may a small change in $c$ lead to a huge change in the sophistication or does it only affect it by a constant factor? In order to better understand this point let us give an example.

**Example 6.4** *By Theorem 4.2.5 we know that for all* $n$ *there is a string* $x \in \Sigma^n$ *such that* $soph_c(x|n) > n - c$. *Note that* $soph_c(x|n) > n - c$ *implies that* $K(x|n) > n - c$ *so this string is* $c$*-random. If we allow some more redundancy, by using a constant* $c_2 > c$ *big enough, we can use the PRINT program in the sophistication.*

*Based on this, it is not true that for any* $c_1$ *used in the sophistication we have that for all* $c_2$ *there is a* $c_3$ *such that*

$$|soph_{c_1}(x|n) - soph_{c_2}(x|n)| < c_3.$$

*As we can always fix an* $c_2$ *that allows us to use the PRINT program, this will lead to a big gap between* $soph_{c_1}$ *and* $soph_{c_2}$.

**Conjecture 6.5** *Sophistication is not stable.*

As we believe that this conjecture is true, we introduced a new variant of sophistication called *coarse sophistication.* Furthermore, we proved the following result regarding coarse sophistication, given $x \in \Sigma^*$ and $O(\log n)$ bits, we can solve the halting problem for all programs $q$ of size smaller than $\frac{csoph(x)}{2} - 2\log n$. This is a very

strong result, and also points towards a strong connection between sophistication and computational depth. Finally, we proved the equivalence between busy beaver computational depth and coarse sophistication, which suggests the significance of these (equivalent) concepts.

The results proved in this work on coarse sophistication and busy beaver computational depth and the citation above, lead us to believe that these new measures deserves further investigation. We believe that these measures can be very useful not only in complexity theory but also in cryptography.
In fact our main motivation in the beginning was to search for an information measure in public key cryptography. However, we did not make such a big detour from our starting motivation. In fact, the current theory of cryptography in the presence of a computationally bounded adversary need to make some assumptions about the existence of hard problems, usually studied in complexity theory. Therefore an important research topic in cryptography is to find the minimal assumptions required to prove the existence of "secure" cryptosystems.

### Applications to Cryptography

Shannon's work [Sha48], outlined a mathematical definition of the notion of information. Briefly, the entropy (complexity) of a random variable $X$ with probability distribution $p_X$ and alphabet $\chi$ is defined as $H(X) = -\sum_{x\in\chi} p_X(x) \log p_X(x)$, meaning that $H(X)$ bits are sufficient to describe $X$ on average. The use of information theory as a measure of security of cipher systems was the starting point of this thesis.

The notion of *perfect security* was introduced by Shannon and means that an adversary does not have any information about some secret. Perfect security has been used for symmetric cryptosystems, consisting of a sender (Alice), a receiver (Bob), a eavesdropper (Eve) and a open channel from Alice to Bob and to Eve. The secret key $Z$ is shared by Alice and Bob only. Alice encrypts the plaintext $X$ using $Z$ resulting in the cryptogram $Y$ that is sent to Bob and can also be seen by Eve. Bob can recover $X$ with $Z$. A cipher in this model is *perfect* if and only if $X$ and $Y$ are independent variables. Vernam's *one time pad* is a perfectly secure cryptosystem.

Later Diffie and Helman [DH76] motivated on Shannon's work proposed a new cryptographic protocol named *public key cryptography.* The main idea is to separate keys for encryption and decryption: a public key known by anybody (used to encrypt) and a

private key known by its owner (used to decrypt). The security of this systems is based on the *infeasability* of an underlying *computational problem*, such as factoring large numbers or computing discrete logarithms. The notion of security of these systems is always conditional on unproven assumption (hardness of some problem).

So we could expect that both theories, information theory and public key cryptography, provide a new cryptographic framework where we could prove security, but, in fact, information theory does not seem to provide a totally satisfactory framework. Shannon's definition of entropy (complexity) is a purely statistical notion, which ignores the computational difficulty of extracting the information.

While Shannon's entropy measures the complexity in a statistical ensemble, Kolmogorov measures the complexity in an individual object. Surprisingly, under some weak restrictions on the distribution $p$, the expectation of $K(x)$, $\sum_x p(x)K(x)$, achieves the minimal average code word length $H(X)$. As entropy is closely related to Kolmogorov complexity, we can replace it and try to prove perfect security. One of the obvious advantages is that by using Kolmogorov complexity we can incorporate the *computational infeasability* in the formulae by using time bounded Kolmogorov complexity.

We believe that computational depth can be useful in public key cryptography; the more immediate connection is the analysis of Zero Knowledge Proofs of Knowledge. In fact in [GMR89] Goldwasser *et al.* where already looking for a measure of useful information in a communication:

*Which communications convey knowledge? Informally, those that transmit the output of an infeasible computation, a computation that we cannot perform ourselves. For example, if A sends to B n random bits, this will be n bits of information. We would say this contains no* **knowledge**, *however, because B could generate random bits by himself. Similarly, the result of any probabilistic polynomial-time computation will not contain any knowledge. With this in mind we would like to derive an upper bound (expressed in bits) for the* **amount of knowledge** *that a polynomially bounded B can extract from a communication (Goldwasser, Micali and Rackoff in [GMR89])*

This measure of *knowledge* is similar to computational depth. The results on this work and the above citation lead us to believe that this line of research is promising and that the concept of computational depth is useful, as a measure of ***useful information***.

# References


[Ade79] L. M. Adelman. Time, space and randomness. Technical Report MIT/LCS/131, Lab. Comp. Science MIT, 1979.

[Adl78] Leonard M. Adleman. Two theorems on random polynomial time. In *IEEE Symposium on Foundations of Computer Science*, pages 75–83, 1978.

[AFV01] L. Antunes, L. Fortnow, and V. Vinodchandran. Computational depth vs. average polynomial time. Technical Report 2001-097, NEC Research Institute, 2001.

[AFvM01] L. Antunes, L. Fortnow, and D. van Melkebeek. Computational depth. In *Proceedings of the 16th IEEE Conference on Computational Complexity*, pages 266–273, New York, 2001. IEEE.

[BDCGL92] S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the theory of average case complexity. *J. Computer System Sci.*, 44(2):193–219, 1992.

[BDG86] J. L. Balcázar, J. Díaz, and J. Gabarró. On non-uniform polynomial space. In Alan L. Selman, editor, *Structure in Complexity Theory.* Springer-Verlag, 1986.

[BDG95] J. Balcazar, J. Diaz, and J. Gabarro. *Structural Complexity 1.* Springer, Berlin, 1995.

[Ben85] C. H. Bennett. Dissipation, information, computational complexity and the definition of organization. In D. Pines, editor, *Emerging Syntheses in Science*, pages 215–233. Addison-Wesley, 1985.

[Ben86] C. H. Bennett. On the nature and origin of complexity in discrete, homogeneous, locally-interacting systems. *Foundations of Physics*, 16:585–592, 1986.

[Ben88] C. H. Bennett. Logical depth and physical complexity. In R. Herken, editor, *The Universal Turing Machine: A Half-Century Survey*, pages 227–257. Oxford University Press, 1988.

[BF97] H. Buhrman and L. Fortnow. Resource-bounded kolmogorov complexity revisited. In *Proceedings of the 14th Symposium on Theoretical Aspects of Computer Science*, pages 105–116, Berlin, 1997. Springer.

[BFL01] H. Buhrman, L. Fortnow, and S. Laplante. Resource-bounded kolmogorov complexity revisited. *SIAM Journal on Computing*, 2001.

[BG81] Charles H. Bennett and John Gill. Relative to a random oracle $A$, $\mathrm{P}^A \neq \mathrm{NP}^A \neq co - \mathrm{NP}^A$ with probability 1. *SIAM J. Comput.*, 10(1):96–113, 1981.

[BG91] A. Blass and Y. Gurevich. On the reduction theory for average-case complexity. In Springer Lecture Notes in Computer Science, editor, *CSL'90, 4th Workshop on Computer Science Logic*, pages 17–30, 1991.

[BGS75] T. Baker, J. Gill, and R. Solovay. Relativizations of the question $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$. *SIAM Journal on Computing*, 4(4):431–442, 1975.

[BLM00] H. Buhrman, S. Laplante, and P. Miltersen. New bounds for the language compression problem. In *Proceedings of the 15th IEEE Conference on Computational Complexity*, New York, 2000. IEEE.

[Cha66] Gregory J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the ACM*, 13(4):145–149, 1966.

[Cha75] Gregory J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22(3):329–340, 1975.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Cov85] T. M. Cover. Kolmogorov complexity, data compression, and inference. In J. K. Skwirzynski, editor, *The Impact of Processing Techniques on Communications*, pages 23–33. Martinus Nijhoff Publishers, 1985.

[Dal82] Robert P. Daley. Busy beaver sets: Characterizations and applications. *Information and Control*, 52:52–67, 1982.

[DH76] W. Diffie and M. Helman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[FK96] L. Fortnow and M. Kummer. On resource-bounded instance complexity. *Theoretical Computer Science*, 161:123–140, 1996.

[FL98] L. Fortnow and S. Laplante. Nearly optimal language compression using extractors. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science*, pages 84–93, Berlin, 1998. Springer.

[For89] Lance Fortnow. *Complexity-theoretic aspects of interactive proof systems.* PhD thesis, Massachusetts Institute of Technology, 1989.

[For01] Lance Fortnow. Kolmogorov complexity. In R. Downey and D. Hirschfeldt, editors, *Aspects of Complexity, Minicourses in Algorithmics, Complexity, and Computational Algebra, Gruyter Series in Logic and Its Applications*, volume 4. Gruyter, 2001.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NPCompleteness.* W.H. Freeman and Company, 1979.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proofs. *SIAM J. on Computing*, 18:186–208, 1989.

[Gol88] O. Goldreich. Towards a theory of average case complexity. http://www.wisdom.weizmann.ac.il/ oded/PS/aver-notes.ps, 1988.

[GTV01] P. Gács, J. Tromp, and P. Vitányi. Algorithmic statistics. *IEEE Trans. Inform. Theory*, 47(6):2443–2463, 2001.

[Gur91] Yuri Gurevich. Average case completeness. *Journal of Computer and System Sciences*, 42(3):346–398, 1991.

[Har83] J. Hartmanis. Generalized kolmogorov complexity and the structure of feasible computations. In *Proceedings, 24 IEEE Symposium on Foundations of Computing*, pages 439–445, 1983.

[HILL99] J. Håstad, R. Impagliazzo, L. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 4(28):1364–1396, 1999.

[HS65] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[Imp95] Russell Impagliazzo. A personal view of average-case complexity. In *Structure in Complexity Theory Conference*, pages 134–147, 1995.

[JL00] David W. Juedes and Jack H. Lutz. Modeling time-bounded prefix kolmogorov complexity. *Theory of Computing Systems*, 33(2):111–123, 2000.

[KA91] M. Koppel and H. Atlan. An almost machine-independent theory of program-length complexity, sophistication, and induction. *Information Sciences*, 56:23–33, 1991.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, pages 85–103. Plenum Press, New York, 1972.

[KL80] R. Karp and R. Lipton. Some connections between nonuniform and uniform complexity classes. In *Conference Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 302–309, 1980.

[Kol65] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems Inform. Transmission*, 1(1):1–7, 1965.

[Kop87] M. Koppel. Complexity, depth, and sophistication. *Complex Systems*, 1:1087–1091, 1987.

[Kop88] M. Koppel. Structure. In R. Herken, editor, *The Universal Turing Machine: A Half-Century Survey*, pages 435–452. Oxford University Press, 1988.

[Lap97] Sophie Laplante. *Kolmogorov Techniques in Computational Complexity Theory.* PhD thesis, University of Chicago, 1997.

[Lau83] Clemens Lautemann. Bpp and the polynomial hierarchy. *Information Processing Letters*, 17(4):215–217, 1983.

[Lev73a] L. Levin. On the notion of a random sequence. *Soviet Math. Dokl*, 14:1413, 1973.

[Lev73b] L. A. Levin. Universal search problems. *Problems Inform. Transmission*, 9:265–266, 1973.

[Lev74] L.A. Levin. Laws of information conservation (non-growth) and aspects of the foundation of probability theory. *Problems Inform. Transmission*, 10:206–210, 1974.

[Lev86] Leonid A. Levin. Average case complete problems. *SIAM J. Comput.*, 15(1):285–286, 1986.

[LM93] Luc Longpré and Sarah Mocas. Symmetry of information and one-way functions. *Information Processing Letters*, 46(2):95–100, 1993.

[LV93] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, 1993.

[LV97a] M. Li and P. Vitányi. Average-case analysis using kolmogorov complexity. In *Advances in Algorithms, Languages, and Complexity*, pages 157–169, 1997.

[LV97b] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, 1997.

[LW95] Luc Longpré and Osamu Watanabe. On symmetry of information and polynomial time invertibility. *Information and Computation*, 121(1):14–22, 1995.

[Mil91] Peter Bro Miltersen. The complexity of malign ensembles. In *Structure in Complexity Theory Conference*, pages 164–171, 1991.

[NW94] N. Nisan and A. Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences*, 49:149–167, 1994.

[Pap85] Papadimitriou. *Computational Complexity.* Wiley, 1985.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[Sch73] C. P. Schnorr. Process complexity and effective random tests. *J. Comp. System Sci.*, 7:376–388, 1973.

[Sch90] Robert E. Schapire. The emerging theory of average-case complexity. Technical Report MIT/LCS/TM-431, MIT Laboratory for Computer Science, 1990.

[Sch99] Rainer Schuler. Universal distributions and time-bounded kolmogorov complexity. In *Proc. 16th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1563, pages 434–443. Springer-Verlag, 1999.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Tech. J.*, 27:379–423; 623–656, 1948.

[She83] A. KH. Shen. The concept of $(\alpha, \beta)$-stochasticity in the kolmogorov sense, and its properties. *Soviet Math. Dokl.*, 28:295–299, 1983.

[SHI65] R. E. Stearns, J. Hartmanis, and P. M. Lewis II. Hierarchies of memory limited computations. In IEEE, editor, *Proceedings of the Sixth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 179–190, 1965.

[Sip83] M. Sipser. A complexity theoretic approach to randomness. In *Proceedings of the 15th ACM Symposium on the Theory of Computing*, pages 330–335, 1983.

[Sip97] M. Sipser. *Introduction to the Theory of Computation.* PWS Publisher, 1997.

[Sol64] R. Solomonoff. A formal theory of inductive inference, part i. *Information and Control*, 7(1):1–22, 1964.

[SY96] R. Schuler and T. Yamakami. Structural average case complexity. *Journal of Computer and Systems Sciences*, pages 308–327, 1996.

[vM99] D. van Melkebeek. *Randomness and Completeness in Computational Complexity.* PhD thesis, The University of Chicago, Department of Computer Science, 1999.

[V'y99] V.V. V'yugin. Algorithmic complexity and stochastic properties of finite binary sequences. *The Computer Journal*, 42(4):294–317, 1999.

[Ž83] Stanislav Žák. A turing machine time hierarchy. *Theoretical Computer Science*, 26(3):327–333, 1983.

[WB95] Jie Wang and Jay Belanger. On the NP-isomorphism problem with respect to random instances. *Journal of Computer and System Sciences*, 50(1):151–164, 1995.

[Yam97] Tomoyuki Yamakami. *Average Case Computational Complexity Theory.* PhD thesis, Department of Computer Science, University of Toronto, 1997.

[ZL70a] A. Zvonkin and L. Levin. The complexity of finite objects and the algorithmic concepts of information and randomness. *Russian Math. Surveys*, 25(6):83–124, 1970.

[ZL70b] A.K. Zvonkin and L.A. Levin. The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Math. Surveys*, 25(6):83–124, 1970.