Une approche expérimentale à la théorie algorithmique de la complexité

Hector Zenil

ORDRE : 40535

UNE APPROCHE EXPÉRIMENTALE À LA THÉORIE ALGORITHMIQUE DE LA COMPLEXITÉ

THÈSE

par

Hector ZENIL

présentée pour obtenir le grade de :

Docteur

Spécialité : ***Informatique***

UNIVERSITÉ DE LILLE 1

Laboratoire d'Informatique Fondamentale de Lille

(UMR CNRS 8022)

JURY :

Cristian S. CALUDE	University of Auckland	*Rapporteur*
Gregory CHAITIN	Universidad de Buenos Aires	*Rapporteur*
Jean-Paul DELAHAYE	Université de Lille 1	*Directeur de thèse*
Serge GRIGORIEFF	Université de Paris 7	*Rapporteur*
Philippe MATHIEU	Université de Lille 1	*Examinateur*
Hervé ZWIRN	Université de Paris 7	*Examinateur*

Table des matières

I Introduction **1**

0.1 De la théorie classique des probabilités à la théorie du calcul . 3

0.1.1 La machine de Turing 4

0.1.2 Le problème du Castor affairé 5

0.1.3 Le contenu d'information d'un objet individuel 5

0.1.4 L'aléatoire mathématique 6

0.1.5 Convergence des définitions 6

0.2 Complexité algorithmique 7

0.2.1 L'aléatoire algorithmique par incompressibilité 8

0.2.2 Le théorème d'invariance 8

0.2.3 Le choix de la machine universelle est important 9

0.2.4 Resumé des propriétés de la complexité algorithmique . 10

0.3 Probabilité algorithmique et distribution universelle 10

0.3.1 Complexité "préfixe" 10

0.3.2 Le nombre Omega (Ω) de Chaitin 11

0.3.3 L'inférence algorithmique de Solomonoff et la semi-mesure de Levin 12

0.3.4 De la métaphore du singe dactylographe de Borel au singe programmateur de Chaitin 13

0.3.5 La profondeur logique de Bennett 14

1 Résumé des chapitres **17**

1.1 Fondements 17

1.1.1 Complexité de Kolmogorov-Chaitin des suites courtes . 17

1.1.2 Évaluation numérique d'une distribution expérimentale de Levin 18

1.2 Applications 21

1.2.1 Recherche des propriétés dynamiques des automates cellulaires et d'autres systèmes par des techniques de compression 21

1.2.2 Classification d'images par complexité organisée : une application de la profondeur logique de Bennett 22

1.2.3 Le fossé de Sloane 25

1.3 Réflexions 26

1.3.1 Sur la nature algorithmique du monde 26

1.3.2 Une approche algorithmique du comportement des marchés financiers 28

1.3.3 Complexité algorithmique versus complexité de temps de calcul : recherche dans l'espace des petites machines de Turing 30

1.4 Résumé des principales contributions 31

II Fondements **37**

2 On the Kolmogorov-Chaitin Complexity for Short Sequences **39**

3 Numerical Evaluation of Algorithmic Complexity for Short Strings **45**

3.1 Introduction 45

3.2 Preliminaries 47

3.2.1 The Halting problem and Chaitin's Ω 47

3.2.2 Algorithmic (program-size) complexity 48

3.2.3 Algorithmic probability 49

3.2.4 The Busy Beaver problem 50

3.3 The empirical distribution D . . . 51
3.4 Methodology . . . 53
3.4.1 Numerical calculation of D . . . 54
3.5 Results . . . 56
3.5.1 Algorithmic probability tables . . . 56
3.5.2 Derivation and calculation of algorithmic complexity . 64
3.5.3 Runtimes investigation . . . 68
3.6 Discussion . . . 70
3.7 Concluding remarks . . . 72

III Applications **75**

4 Compression-Based Investigation of the Dynamical Properties of Cellular Automata and Other Systems **77**
4.1 Introduction . . . 77
4.2 Preliminaries . . . 78
4.3 Compression-based classification . . . 80
4.3.1 Compression-based classification of elementary cellular automata from simplest initial conditions . . . 80
4.4 Compression-based clustering . . . 82
4.4.1 2-clusters plot . . . 82
4.4.2 Compression-based classification of larger spaces of cellular automata and other abstract machines . . . 86
4.5 Compression-based phase transition detection . . . 89
4.5.1 Initial configuration numbering scheme . . . 89
4.5.2 Phase transitions . . . 92
4.6 Compression-based numerical computation of a characteristic exponent . . . 93
4.6.1 Regression analysis . . . 96
4.6.2 Phase transition classification . . . 98
4.7 Conclusion . . . 102

5 Image Characterization and Classification by Physical Complexity 105

5.1 Introduction . . . 105
5.2 Theoretical basis . . . 106
5.2.1 Algorithmic complexity . . . 106
5.2.2 Bennett's logical depth . . . 108
5.3 Methodology . . . 111
5.3.1 Towards the choice of lossless compression algorithm . 114
5.3.2 Compression method . . . 115
5.3.3 Using decompression times to estimate complexity . . . 116
5.3.4 Timing method . . . 117
5.4 Results . . . 118
5.4.1 Controlled experiments . . . 118
5.4.2 Calibration tests . . . 123
5.4.3 Compression length ranking . . . 125
5.4.4 Decompression times ranking . . . 125
5.5 Conclusions and further work . . . 130

6 Sloane's Gap : Do Mathematical and Social Factors Explain the Distribution of Numbers in the OEIS ? 135

6.1 Introduction . . . 135
6.2 Presentation of the database . . . 136
6.3 Description of the cloud . . . 140
6.3.1 General shape . . . 140
6.3.2 Defining the gap . . . 140
6.3.3 Characteristics of numbers "above" . . . 141
6.4 Explanation of the cloud-shape formation . . . 144
6.4.1 Overview of the theory of algorithmic complexity . . . 144
6.4.2 The gap: A social effect? . . . 147
6.5 Conclusion . . . 149

IV Réflexions **151**

7 On the Algorithmic Nature of the World **153**

7.1 Introduction . . . 153
7.1.1 Levin's universal distribution . . . 154
7.2 The null hypothesis . . . 155
7.2.1 Frequency distributions . . . 156
7.2.2 Computing abstract machines . . . 156
7.2.3 Physical sources . . . 160
7.2.4 Hypothesis testing . . . 162
7.2.5 The problem of overfitting . . . 163
7.3 Possible applications . . . 167
7.4 The meaning of *algorithmic* . . . 169
7.5 Conclusions . . . 171

8 An Algorithmic Information-theoretic Approach to the Behavior of Financial Markets **175**

8.1 Introduction . . . 175
8.2 Preliminaries . . . 176
8.3 The traditional stochastic approach . . . 177
8.4 Apparent randomness in financial markets . . . 179
8.5 An information-theoretic approach . . . 182
8.5.1 Algorithmic complexity . . . 184
8.5.2 Algorithmic probability . . . 185
8.6 The study of the real time series v. the simulation of an algorithmic market . . . 187
8.6.1 From AIT back to the behavior of financial markets . . . 188
8.6.2 Unveiling the machinery . . . 189
8.6.3 Binary encoding of the market direction of prices . . . 192
8.6.4 Calculating the algorithmic time series . . . 194
8.7 Experiments and Results . . . 195
8.7.1 Rankings of price variations . . . 195

8.7.2 Backtesting . . . 199
8.7.3 Algorithmic inference of rounded price directions . . . 200
8.8 Further considerations . . . 201
8.8.1 Rule-based agents . . . 201
8.8.2 The problem of over-fitting . . . 202
8.9 Conclusions and further work . . . 204

9 Program-size Versus Time Complexity 209
9.1 Introduction . . . 209
9.1.1 Two measures of complexity . . . 210
9.1.2 Turing machines . . . 211
9.1.3 Relating notions of complexity . . . 212
9.1.4 Investigating the micro-cosmos of small Turing machines 213
9.1.5 Plan of the paper . . . 213
9.2 Methodology and description of the experiment . . . 214
9.2.1 Methodology in short . . . 214
9.2.2 Resources . . . 215
9.2.3 One-sided Turing Machines . . . 215
9.2.4 Unary input representation . . . 216
9.2.5 Binary output convention . . . 217
9.2.6 The Halting Problem and Rice's theorem . . . 217
9.2.7 Cleansing the data . . . 218
9.2.8 Running the experiment . . . 221
9.3 Investigating the space of 2-state, 2-color Turing machines . . . 221
9.3.1 Determinant initial segments . . . 222
9.3.2 Halting probability . . . 222
9.3.3 Phase transitions in the halting probability distribution 223
9.3.4 Runtimes . . . 225
9.3.5 Clustering in runtimes and space-usages . . . 226
9.3.6 Definable sets . . . 228
9.3.7 Clustering per function . . . 228

9.3.8 Computational figures reflecting the number of available resources 228
9.3.9 Types of computations in (2,2) 229
9.4 Investigating the space of 3-states, 2-colors Turing machines . 232
9.4.1 Determinant initial segments 232
9.4.2 Halting probability 232
9.4.3 Runtimes and space-usages 233
9.4.4 Definable sets 234
9.4.5 Clustering per function 234
9.4.6 Exponential behavior in (3,2) computations 235
9.5 The space (4,2) 235
9.6 Comparison between the TM spaces 236
9.6.1 Runtimes comparison 236
9.6.2 Distributions over the complexity classes 239
9.6.3 Quantifying the linear speed-up factor 240

Remerciements

Je tiens avant tout à remercier mon directeur de thèse Jean-Paul Delahaye qui m'a guidé et encouragé a entreprendre cette belle aventure. À mes parents, ma famille, mes amis et colleagues. Aux membres du jury de thèse Cris Calude, Greg Chaitin, Serge Grigorieff, Philippe Mathieu et Hervé Zwirn. D'une manière spéciale à Cris Calude, Greg Chaitin et Stephen Wolfram pour leurs conseils et appuis permanents. Je voudrais aussi remercier Jean Mosconi, Bernard François, Jacques Dubucs, Matthew Szudzik et Todd Rowland. Aussi à Joost Joosten, Tommaso Bolognesi, Fernando Soler-Toscano, Genaro Juárez Martínez, Barry Cooper, Clément Vidal, Wilfried Sieg, Kevin Kelly, Klaus Sutner et Jeremy Avigad, pour les discussions qui ont été un stimulant continu.

Acknowledgments

I'd like to begin by acknowledging my thesis advisor, Jean-Paul Delahaye, who has guided and encouraged me along this felicitous adventure. Heartfelt thanks to my parents, family, friends and colleagues. To the members of the evaluation committee : Cris Calude, Greg Chaitin, Serge Grigorieff, Philippe Mathieu and Hervé Zwirn. Special thanks to à Cris Calude, Greg Chaitin and Stephen Wolfram for their constant support and helpful advice. I would also like to thank Jean Mosconi, Bernard François, Jacques Dubucs, Matthew Szudzik and Todd Rowland. And thanks to à Joost Joosten, Tommaso Bolognesi, Fernando Soler-Toscano, Genaro Juárez Martínez, Barry Cooper, Clément Vidal, Wilfried Sieg, Kevin Kelly, Klaus Sutner and Jeremy Avigad for richly stimulating discussions and for their support.

Agradecimientos

Quisiera antes que nada agradecer a Jean-Paul Delahaye quien me guió y motivó durante el desarrollo de esta bella aventura. A mis padres, mi familia, amigos y colegas. A los miembros del jurado Cris Calude, Greg Chaitin, Serge Grigorieff, Philippe Mathieu y Hervé Zwirn. Particularment a Cris Calude, Greg Chaitin y Stephen Wolfram por sus consejos y apoyo permanente. Quisiera también agradecer a Jean Mosconi, Bernard François, Jacques Dubucs, Matthew Szudzik y Todd Rowland. Y también a Joost Joosten, Tommaso Bolognesi, Fernando Soler-Toscano, Genaro Juárez Martínez, Barry Cooper, Clément Vidal, Wilfried Sieg, Kevin Kelly, Klaus Sutner y Jeremy Avigad, por las enriquecedoras discusiones y su apoyo.

Première partie

Introduction

Brève introduction à la théorie du calcul et à la théorie algorithmique de l'information

Cette thèse est une contribution à la recherche des applications de la théorie algorithmique de l'information. Il s'agit également d'utiliser l'aléatoire comme concept de recherche pour appliquer la théorie de l'information au monde réel et pour fournir des explications vraisemblables à des phénomènes du monde physique.

Commençons par illustrer les motivations et les concepts de cette théorie qui vont nous accompagner tout au long de cette thèse.

0.1 De la théorie classique des probabilités à la théorie du calcul

Imaginons que nous nous trouvions face à une suite 01010101010101010101010101010101. De prime abord, nous pouvons penser que cette suite n'est pas aléatoire. Cependant, cette suite a la même probabilité d'occurrence que n'importe quelle autre suite de la même longueur. Le fait de douter de sa nature aléatoire n'est qu'un préjugé selon la théorie classique des probabilités.

Sous une distribution uniforme, toutes les suites s à n chiffres (par exemple pour des suites qui encodent les résultats d'un jeu de pile ou face avec une pièce non truquée) :

00000000000000000000000000000000
01010101010101010101010101010101
11001100101101001011011110011 0

ont toutes la même probabilité $Pr(s) = 1/2^n$ comme résultat de lancer la pièce n fois (0 = pile, 1 = face). Par contre, si on obtient la première, on supposera que la pièce était truquée de même si on tombe sur la deuxième, tandis que si on tombe sur la troisième, cela semblera normal.

Malgré tout, notre intuition persiste à dire qu'une suite telle que 000000... ou 111111... n'est pas aléatoire. La théorie algorithmique de l'information donne une explication à cette intuition par le biais du concept de contenu d'information. On voudrait formaliser l'idée qu'on n'attend pas de régularités d'un processus dans une suite d'évènements résultant du hasard, et que chaque suite est capable de contenir de l'information pour les différencier.

Les définitions que la théorie algorithmique de l'information fournit sont des définitions de type négatif, c'est-à-dire qu'on donne une définition en termes de calculabilité de l'objet non aléatoire, puis on définit un objet aléatoire comme étant un objet qui ne possède pas cette propriété même si on ne le montre pas.

La théorie de l'aléatoire algorithmique se base sur le modèle de calcul de Turing. La thèse de Church-Turing, largement acceptée comme vraisemblable, affirme qu'est "calculable" ce qui est "calculable par une machine de Turing" (ou tout autre modèle de calcul équivalent). On dit alors que les fonctions qu'on peut calculer avec une machine de Turing ou avec un algorithme sont les fonctions calculables (on dit aussi récursives). Les fonctions usuelles (n $\rightarrow n^2$, n $\rightarrow$ nième nombre premier, etc.) sont calculables.

0.1.1 La machine de Turing

Le modèle de Turing[18] est un modèle abstrait d'ordinateur composé d'un mécanisme de calcul et d'un ruban sur lequel la machine écrit et qu'elle peut effacer en déplaçant une tête de lecture-écriture. Chaque machine de Turing est associée à un programme qui détermine avec une précision absolue les opérations qu'elle effectue. Alan Turing démontre qu'il existe un type de machine de Turing (dite universelle) capable de simuler toute autre machine de Turing.

Le problème de l'arrêt

On démontre qu'il existe des fonctions non calculables : la première d'entre elles est la fonction qui, pour tout programme donné P, indique (a) si le programme finit par s'arrêter ou (b) s'il continue indéfiniment à calculer. Effectivement, Turing démontre que cette fonction n'est pas calculable, c'est

ce qu'on nomme le problème de l'indécidabilité de l'arrêt. En d'autres termes, il n'existe aucune machine de Turing qui sait déterminer si une autre machine va s'arrêter pour un programme avec une entrée quelconque. Beaucoup d'autres problèmes sur les machines de Turing ne sont pas décidables.

La pertinence du langage binaire

Le langage binaire permet d'encoder des événements comme ceux du pile ou face mais aussi toute autre série d'événements discrets par le biais d'une transformation de base. Par exemple, si on étudie les sorties d'un dé qui peut donner six résultats possibles, on peut les encoder chacun par un symbole différent et les transformer en binaire.

0.1.2 Le problème du Castor affairé

On dénomme Castor affairé[**?**] à n états, la machine de Turing à n états qui calcule le plus longtemps ou produit le plus de 1 sur le ruban avant de s'arrêter. Le temps maximal se note $S(n)$ et le nombre maximal de 1 sur le ruban se note $\Sigma(n)$. Le Castor affairé à 3 états calcule durant $S(3) = 21$ étapes, et donc, toute machine ayant trois états calcule moins de 22 étapes. Le Castor affairé à 4 états calcule durant $S(4) = 107$ étapes. Le Castor affairé à 5 étapes n'a pas encore été identifié avec certitude, mais on pense que $S(5) = 47\,176\,870$. Les foncions $n \to S(n)$, $n \to \Sigma(n)$ sont non calculables (à cause de l'indecidabilité de l'arrêt des machines de Turing).

0.1.3 Le contenu d'information d'un objet individuel

Grâce au modèle de calcul de Turing, on a un outil puissant pour formaliser le concept de complexité d'une suite par le biais de son contenu d'information.

Pour être prudent et encadrer cette nouvelle théorie de contenu d'information, il faut adopter le concept de calcul pour décrire un objet par un "programme informatique" que produit l'objet (ou sa représentation).

La notion de calculabilité n'étant pas connue auparavant, il a été difficile de donner un cadre stable à la définition algorithmique de l'aléatoire, mais l'utilisation d'un modèle de calcul a donné lieu à plusieurs formalisations.

La solution au phénomène des sorties à pile ou face, dont nous avons parlé dans la première section, a été proposée indépendamment, dans les

années 60, par Solomonoff[17], Kolmogorov[10] et Chaitin[4] (ce deuxième déjà à l'origine de la formalisation axiomatique du calcul des probabilités). Si la suite de 000... (0 répété 1 million de fois) nous semble surprenante comme produit du hasard, c'est parce qu'elle est très facile à décrire et non pas difficile comme on l'attendrait pour une suite aléatoire.

0.1.4 L'aléatoire mathématique

Ce n'est qu'avec l'arrivée des travaux de Schnorr et Martin-Löf que la définition de l'aléatoire mathématique prend sa forme actuelle par le biais du concept de martingale[16] (stratégie de pari) et de test statistique effectif[13]. On peut classer les définitions en trois catégories : typicalité, imprédictibilité et incompressibilité.

La notion de typicalité est basée sur la théorie de la mesure, elle formalise l'idée intuitive qu'une séquence binaire infinie satisfait toutes les propriétés statistiques pouvant être testées de façon algorithmique. Une suite est aléatoire au sens de Martin-Löf[13], si et seulement si, s a toutes les propriétés communes à la plupart des séquences binaires.

La notion d'imprédictibilité exprime le fait qu'une séquence binaire est aléatoire s'il n'existe aucune façon algorithmique d'en prédire ses bits. Une suite est aléatoire au sens de Schnorr[16] si elle est imprévisible et donc aucune stratégie de pari effective ne peut mener à un gain à long terme si l'on parie sur les bits de la séquence. En d'autres mots, une séquence infinie aléatoire doit être une séquence ne possédant aucune structure, régularité, ou règle de prédiction identifiable.

D'autre part, la notion de compressibilité permet de définir la complexité d'une suite infinie. Une séquence est aléatoire au sens de Chaitin-Levin[4, 11] si tout segment initial de la séquence est incompressible. La section suivante fournira plus de détails.

0.1.5 Convergence des définitions

Ces notions de l'aléatoire caractérisent aujourd'hui mathématiquement ce qu'on entend par séquence aléatoire. Shnorr montre qu'une séquence est non aléatoire s'il existe une stratégie calculable permettant de gagner de l'argent sous certaines conditions. Une variante de la notion de prédictibilité selon Shnorr[16] produit la même classe de suites aléatoires qu'au sens de Martin-Löf et de Chaitin-Levin.

Le fait que les notions se révèlent mathématiquement équivalentes[4, 16] alors qu'elles sont fondées sur des notions différentes et conçues indépendamment les unes des autres, conduit à penser que ces définitions réussissent à saisir de façon fondamentale le concept de l'aléatoire.

0.2 Complexité algorithmique

La définition de l'aléatoire donnée par la complexité algorithmique fournit des outils pratiques qui permettent son application aux suites finies, à la différence des définitions abstraites et infinitaires de Martin-Löf[13] ou Schnorr[16].

La complexité algorithmique, introduite indépendamment et presque simultanément par Solomonoff, Kolmogorov, Chaitin et Levin, définit la complexité d'une suite comme la taille du plus court programme qui la génère. La complexité algorithmique d'une suite est alors comprise entre 0 et (à peu près) sa longueur. Plus formellement,

Définition 1 [10, 4, 17, 11] : La complexité algorithmique de $s \in \{0,1\}^*$, relativement à une machine de Turing universelle U, est définie par

$$C_U(s) = min\{|p| : U(p) = s\}$$

où $|p|$ désigne la longueur de p.

Donc, en utilisant la complexité algorithmique, on peut maintenant déterminer que la suite :

00000000000000000000000000000000000000000000000000

a une complexité algorithmique plus faible que la suite :

11101001001101110101101001001100011010000001101000

La première peut être produite par un programme du type "imprime 50 zéros" qui est vraisemblablement plus court qu'un programme nécessaire pour créer la deuxième suite.

0.2.1 L'aléatoire algorithmique par incompressibilité

Définition 2. : Une suite binaire finie s est dite aléatoire si $C(s)$ est proche de la longueur originale de s, autrement dit si s n'est pas compressible. De même, pour un type d'objet fini, on dit qu'un objet x est aléatoire si sa représentation sous forme de suite binaire $code(x)$ l'est (l'encodage étant fixé à l'avance).

Parmi les propriétés des suites binaires, il y a notamment le fait que la majorité des suites finies ont une complexité maximale (c'est-à-dire proche de leur longueur). Par un argument combinatoire, une suite tirée au hasard a toutes les chances d'avoir une complexité algorithmique maximale dès que n est assez grand. Par exemple, parmi toutes les suites de 0 et de 1 de longueur n, pour n fixé :

- moins d'une suite sur 1 024 a une complexité $< n-10$, c'est-à-dire peut être comprimée de plus de 10 digits ;
- moins d'une suite sur un million a une complexité $< n-20$, c'est-à-dire peut être comprimée de plus de 20 digits, etc.

L'une des propriétés les plus importantes de cette mesure (pour l'une ou l'autre version) est qu'aucune des deux n'est calculable, c'est-à-dire, qu'il n'existe pas d'algorithme pour calculer la complexité d'une suite quelconque, car aucune procédure ne garantit de trouver le programme le plus court qui l'engendre. Ceci fait douter de trouver des applications pratiques.

Des approximations à ce concept sont, cependant, envisageables. Une façon de contourner le problème de calculabilité est de considérer des approximations calculables. La longueur de la plus courte "description" peut être approchée par sa plus courte forme compressée, en utilisant un algorithme de compression fixé qui repère des régularités.

Il n'y a pas, en général, de méthode effective pour trouver la meilleure compression, mais toute compression possible donne une borne supérieure, et donc la taille de la suite compressée est une approximation à sa complexité algorithmique. Cette approche est utilisée pour des applications pratiques, notamment par Cilibrasi et Vitányi[12] pour la classification de données.

0.2.2 Le théorème d'invariance

On peut se demander si la complexité algorithmique est stable lorsqu'on change de langage de base, par exemple de machine universelle de Turing U.

Est-ce qu'en changeant de langage (ou de machine universelle de Turing), on ne change pas la complexité mesurée ?

La définition de la complexité de Kolmogorov n'est pas absolument indépendante de la machine U, mais elle l'est à une constante prés.

Le théorème d'invariance donne sens à la définition de la complexité algorithmique :

Théorème (invariance[17]) : Si L et M sont deux machines de Turing universelles, et si on note $C_L(s)$ et $C_M(s)$ la complexité algorithmique quand on utilise L ou M comme machine de référence, alors il existe une constante $c_{L,M}$ telle que pour toute suite binaire finie s :

$$|C_L(s) - C_M(s)| < c_{L,M}$$

Ce théorème se demontre en utilisant la possibilité d'écrire en L un compilateur pour M, et réciproquement.

0.2.3 Le choix de la machine universelle est important

Alors que la formalisation de la notion d'algorithme dans la théorie de la calculabilité a permis de fixer le modèle de calcul sur lequel on peut définir la complexité algorithmique d'une suite (avec une machine universelle de Turing), sans la relativiser à aucun autre modèle de calcul (ce qui a permis d'encadrer la notion de complexité), la dépendance du langage joue un rôle important lorsqu'on veut évaluer la complexité d'une suite finie, comme le théorème d'invariance le met en évidence.

Pour les suites courtes par exemple, parler de $C(s)$ ne semble pas avoir vraiment de sens, car la constante additive c est trop grande par rapport à la taille des suites courtes. C'est cette constante (le résultat du changement de cadre de référence) qui va être le sujet de réflexion des premiers chapitres de cette thèse. On suggère une manière de contourner expérimentalement cette difficulté par une méthode où la constante additive devient moins importante, voire négligeable, particulierment pour les suites courtes. Au moyen d'un calcul massif, nous produirons des distributions de fréquences pour évaluer la probabilité algorithmique d'une suite et ensuite sa complexité. Le but : fournir une méthode alternative aux algorithmes de compression traditionnellement utilisés (mais inutiles pour de suites courtes) et rendre plus stable la définition de complexité algorithmique.

Nous allons ainsi, montrer que de vraies applications de la théorie algorithmique de l'information sont envisageables, de l'utilisation du concept de probabilité d'arrêt Ω de Chaitin, en passant par la mesure de Levin, jusqu'à la profondeur logique de Bennett dont on parlera dans les prochaines sections.

0.2.4 Resumé des propriétés de la complexité algorithmique

- La fonction $s \to C(s)$ n'est pas calculable. Aucun algorithme ne peut, pour toute suite s dont on fournit les données, calculer en un temps fini la valeur de $C(s)$.
- En pratique, pour évaluer $C(s)$, on utilise des compresseurs (sans perte) : la taille du fichier comprimé de s par un algorithme de compression (sans perte) est alors une valeur approchée (une borne supérieure) de $C(s)$.
- La non-calculabilité de $C(s)$ a pour conséquence qu'on ne peut jamais être certain d'être proche de sa valeur (car, par exemple, une régularité non vue par le compresseur utilisé peut être présente dans s).
- $C(s)$ est robuste au sens où il existe une équivalence à une constante près qui ne dépend pas de l'objet à mesurer, mais de la machine qu'on utilise pour le mesurer. Cette propriété qui donne sens à la mesure est aussi une contrainte pour évaluer vraiment (numériquement) la complexité d'une suite, car cela dépend de la machine choisie, le théorème d'invariance ne garantissant que la convergence asymptotique.

0.3 Probabilité algorithmique et distribution universelle

0.3.1 Complexité "préfixe"

Étant donné que la complexité algorithmique n'est pas suffisante pour définir rigoureusement certains concepts pour lesquels on parlera de choisir des programmes au hasard, il faut introduire une variation de la complexité algorithmique, il s'agit de la complexité dite "préfixe".

Un code préfixe (aussi appelé comme *code instantané*) est un code ayant la particularité de ne posséder aucun mot ayant pour préfixe un autre mot.

En d'autres termes, aucun mot finissant un code préfixe ne peut se prolonger pour donner un autre mot.

Cette propriété est souvent recherchée pour les codes à longueur variable, par exemple une suite d'instructions de calcul (par exemple un programme), afin de pouvoir les décoder lorsque plusieurs programmes sont concaténés les uns aux autres. Un code préfixe est un code non ambigu. Par exemple, les codes à taille fixe sont tous des codes préfixes. Tel est le cas des numéros de téléphone.

Une mesure adéquate doit être fondée sur des programmes autodélimités[4, 11]. C'est-à-dire où il n'est pas possible de concaténer deux programmes, ce qui rendrait inopérant toute tentative de définition d'une mesure de probabilité, car tout programme qui commence avec un autre programme finirait par contribuer un nombre infini de fois à la probabilité du premier programme, ce qui rendrait la probabilité des programmes divergente et la somme des probabilités de tous les programmes strictement supérieur à 1. On s'arrange donc pour que chaque programme ne soit jamais le début d'un autre (par exemple, si 0001110 est un programme alors 000111011 n'en sera pas un). Un programme délimité est donc un programme qui ne peut pas faire partie d'un autre, car il est limité, par exemple, par une instruction qui signale la fin du programme. Dans plusieurs langages de programmation on utilise des délimitateurs. Par exemple, on dénote la fin des instructions par point-virgule, accolades ou le mot *end*.

Pour tirer un programme au hasard, on procède de la manière suivante : on convient d'écrire les programmes en langage binaire et on oblige chaque programme à posséder une suite unique pour délimiter la fin du programme. Le tirage au hasard d'un programme consiste à choisir, par pile ou face avec une pièce non truquée, des 0 et des 1 jusqu'à avoir un programme complet, et ensuite à le faire fonctionner. Si la suite de tirages de 0 et de 1 ne donne jamais un programme complet, on considère qu'il n'y a pas arrêt.

0.3.2 Le nombre Omega (Ω) de Chaitin

Une machine universelle, c'est-à-dire susceptible de représenter toute fonction calculable, étant donnée, on numérote les programmes par ordre de longueur, puis par ordre "alphabétique" (0 puis 1) à l'intérieur des programmes de même longueur. Soit $p_0, p_1, \ldots, p_n, \ldots$ une telle numérotation.

Chaque programme, une fois lancé, finit par s'arrêter ou, au contraire, poursuit indéfiniment ses calculs (par exemple, parce qu'il boucle).

La définition du nombre Ω de Chaitin fait intervenir le problème de l'arrêt de la manière suivante.

Définition 3 [4] : Ω est la probabilité qu'un programme p tiré au hasard s'arrête : $\Omega = \sum_{\text{p s'arrête}} 2^{-|p|\cdot}$ avec p un programme autodélimité.

De la définition de Ω, on sait tirer toutes sortes d'informations et démontrer ses propriétés (transcendant, équiréparti, etc.). Mais justement, l'une de ces propriétés signifie qu'on ne pourra jamais connaître tous ses bits, qui se comportent comme une suite de tirages aléatoires totalement imprévisibles.

Parmi toutes les propriétés connues du nombre Ω de Chaitin, celles qui nous intéressent dans ce travail sont :

- Le nombre Ω est non calculable, comme la complexité algorithmique, car aucun algorithme ne peut égrainer les chiffres de Ω un par un (toujours à cause du problème de l'arrêt).
- Le nombre Ω est aléatoire dans le sens suivant : le plus court programme qui engendre ses n premiers bits possède environ n bits. Aucun des nombres transcendants classiques ne possède cette propriété. D'ailleurs, les constantes π et e sont facilement compressibles puisqu'on connaît des algorithmes qui en calculent tous les chiffres un par un.
- La connaissance de m bits de Ω permet de savoir pour tout programme p dont la longueur est inférieure à m, s'il s'arrête ou non. La méthode consiste à énumérer tous les programmes en les faisant fonctionner à tour de rôle.

0.3.3 L'inférence algorithmique de Solomonoff et la semi-mesure de Levin

Solomonoff[17] introduit un concept d'inférence algorithmique que Levin[11] formalise avec la semi-mesure [1] qu'il développe mathématiquement dans son concept de recherche universelle. La mesure m (qualifiée de distribution universelle miraculeuse par Ming Li et Walter Kirchherr[9]) indique que plus une suite s est simple, plus sa probabilité $m(s)$ comme résultat d'un calcul, est grand. Plus formellement :

1. *semi* indique que la mesure n'est qu'à moitié calculable. Autrement dit, on peut seulement s'approcher de $m(s)$, car on ne peut pas vraiment la calculer et donc la considérer comme une mesure pleine.

Définition 4 [11] $m(s) = \sum_{\text{p s'arrête et produit la suite s}} 2^{-|p|}$ avec p un programme autodélimité.

Plus une suite s est aléatoire, plus sa probabilité $m(s)$ est petite. $m(s)$ est une distribution de fréquences qui donne la probabilité pour une machine de Turing M de produire une séquence s avec un programme aléatoire. Cette distribution est liée à la complexité algorithmique $C(s)$, car $m(s) = 1/2^{C(s)+O(1)}$[11].

Cette approche formalise le principe connu du Rasoir d'Occam, couramment interprété comme la recherche de l'explication la plus simple pour une observation. Ici l'objet observé est la suite s, et sa plus simple "explication" est le plus petit programme informatique qui produit s.

0.3.4 De la métaphore du singe dactylographe de Borel au singe programmateur de Chaitin

Avec assez de temps, un singe qui tape (on suppose que chaque événement est indépendant, c'est-à-dire que le singe tape vraiment de manière désordonnée) indéfiniment sur une machine à écrire, dactylographiera n'importe quel texte (par exemple le roman "Les Misérables"). La probabilité d'obtenir Les Misérables serait donc $1/50^n$, avec n la taille de l'œuvre de Victor Hugo et 50 le nombre de touches de la machine à écrire.

La probabilité $m(s)$ de produire le roman des Misérables en remplaçant la machine à écrire par un ordinateur est beaucoup plus forte (même si elle reste toujours petite du fait de la longueur du texte), car la complexité $C(s)$ du texte est assez faible par rapport à une suite aléatoire de la même taille.

Autrement dit, le singe a beaucoup plus de chances de produire un roman en tapant un programme sur le clavier d'un ordinateur que sur une machine à écrire. Le singe, n'est qu'une source de programmes aléatoires. La métaphore, formalisée par le concept de probabilité algorithmique (ou semi-mesure de Levin), indique que si on tire des programmes au hasard, on verra que la plupart de ces programmes ne produisent pas de séquences aléatoires mais plutôt organisées.

0.3.5 La profondeur logique de Bennett

Complexe peut signifier aléatoire ou fortement structuré, riche en information. La complexité algorithmique, par exemple, classera les objets suivants de moins à plus complexes :

- un cristal, objet répétitif.
- un ordinateur, un être vivant.
- un gaz, un nuage.

Pourtant, c'est sur la ligne 2 qu'on trouve les objets les plus organisés. C'est la critique formulée par Ilya Prigogine[15] au sujet de la complexité algorithmique comme mesure inadéquate de richesse en organisation.

Une tentative pour définir cette *complexité organisée* est la "profondeur logique" de Charles Bennett[3].

Définition 5 [3] : La profondeur logique d'une suite s est définie par :

$$P(s) = \text{temps de calcul du programme minimal de } s.$$

Ce qui tient compte à la fois de la complexité algorithmique (la taille du programme qui engendre s) et du nombre de pas du calcul que ce programme prend pour engendrer s. La profondeur logique tente de mesurer le "contenu en calcul" d'un objet fini.

Une série d'arguments variés montre qu'une forte profondeur logique est la marque que contient un objet attestant qu'il est le résultat d'un long processus d'élaboration. Le temps de fonctionnement du programme minimal qui correspond à la complexité organisée d'une suite s'oppose à la complexité aléatoire définie par la complexité algorithmique.

Pour en savoir plus

Pour ce qui est de la complexité algorithmique, les livres de référence sont ceux de Calude[5] et de Li et Vitanyi[12]. Pour les liens entre calculabilité, complexité et aléatoire, on peut consulter le livre de Nies[14], ainsi que celui de Downey et Hirschfeldt[8]. Signalons également les introductions à la complexité algorithmique de Jean-Paul Delahaye[6, 7], la thèse de Laurent Bienvenu[1] et son introduction historique[2], ainsi que le livre que j'ai edité[19]. Évidemment, les œuvres séminales de Chaitin[4], Kolmogorov[10], Levin[11], Solomonoff[17], Martin-Löf[13], Schnorr[16] et Bennett[3], restent les textes fondateurs et les références les plus importantes de cette thèse.

Bibliographie

[1] L. Bienvenu. Caractérisations de l'aleatoire par les jeux : impredictabilité et stochasticité. *Thése présentée pour obtenir le grade de docteur*, Université de Provence, 2008.

[2] L. Bienvenu. On the history of martingales in the study of randomness. *journal Electronique d'Histoire des Probabilités et de la Statistique*, vol. 5 No. 1, 2009.

[3] C.H. Bennett. Logical Depth and Physical Complexity. in Rolf Herken (ed) *The Universal Turing Machine–a Half-Century Survey,* Oxford University Press 227-257, 1988.

[4] G. J. Chaitin. *On the length of programs for computing finite binary sequences : Statistical considerations.* Journal of the ACM, 16(1) : 145–159, 1969.

[5] C. S. Calude. *Information and Randomness : An Algorithmic Perspective.* Springer, 2e éd., 2002.

[6] J.-P. Delahaye. *Information, complexité et hasard.* Hermes Sciences Publicat., Édition : 2e éd. revue, 1999.

[7] J.-P. Delahaye. *Complexité aléatoire et complexité organisée.* Editions Quae, 2009.

[8] R. Downey et D. R. Hirschfeldt. Algorithmic Randomness and Complexity. Springer, 2007.

[9] W. Kirchherr, M. Li et P.M.B. Vitányi. The Miraculous Universal Distribution. *Mathematical Intelligencer,* 19 : 4, 7–15. 1997.

[10] A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information and Transmission,* 1(1) : 1–7, 1965.

[11] L. Levin. Universal search problems. *Problems of Information Transmission*, 9 (3) : 265-266, 1973.

[12] M. Li et P.M.B. Vitányi. *An introduction to Kolmogorov complexity and its applications.* Springer, New York, 2e éd. 1997, and 3e éd. 2008.

[13] P. Martin-Löf. The definition of random sequences. *Information and Control*, 9 : 602–619, 1966.

[14] A. Nies. *Computability and Randomness.* Oxford University Press. 2009.

[15] G. Nicolis, I. Prigogine. *Exploring complexity : An introduction.* New York, NY : W. H. Freeman, 1989.

[16] C.-P. Schnorr. *Zufälligkeit und Wahrscheinlichkeit. Eine algorithmische Begründung der Wahrscheinlichkeitstheorie.* Springer, 1971.

[17] R.J. Solomonoff. A formal theory of inductive inference : Parts 1 and 2. Information and Control, 7 : 1–22 and 224–254, 1964.

[18] A.M. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society.* 2 42 : 230–65, 1936, published in 1937.

[19] H. Zenil. (ed.) *Randomness Through Computation.* World Scientific, 2011.

Chapitre 1

Résumé des chapitres

La thèse est divisée en 3 grandes parties : fondements, applications et réflexions. Chaque partie contient plusieurs articles publiés (plus de la moitié) ou actuellement soumis à des revues à comité de lecture et presentés comme chapitres dans cette thèse. La section 1.4 contient une liste chronologique et les références exactes des papiers. Voici un résumé des parties et chapitres.

1.1 Fondements

1.1.1 Complexité de Kolmogorov-Chaitin des suites courtes

Une caractéristique ennuyeuse de la complexité de Kolmogorov-Chaitin (dénotée dans ce chapitre par K) est qu'elle n'est pas calculable, ce qui limite son domaine d'application. Une autre critique concerne la dépendance de K à un langage particulier ou une machine de Turing universelle particulière, surtout pour les suites courtes (plus courtes que les longueurs typiques des compilateurs des langages de programmation).

En pratique, on peut obtenir une approximation de $K(s)$, grâce aux méthodes de compression. Mais les performances de ces méthodes de compression s'écroulent quand il s'agit des suites courtes. Pour les suites courtes, approcher $K(s)$ par des méthodes de compression ne fonctionne pas.

On présente dans ce chapitre une approche empirique permettant de surmonter ce problème. Nous allons proposer une méthode "naturelle" qui donnera une définition stable de la complexité de Kolmogorov-Chaitin $K(s)$ via

la mesure de probabilité algorithmique $m(s)$. L'idée est de faire fonctionner une machine universelle en lui donnant des programmes au hasard pour calculer expérimentalement la probabilité $m(s)$ (la probabilité de produire s), pour ensuite évaluer numériquement $K(s)$. Cette méthode remplacera la méthode des algorithmes de compression. La méthode consiste à : (a) faire fonctionner des mécanismes de calcul (machines de Turing, automates cellulaires) de façon systématique pour produire des suites (b) observer quelles sont les distributions de probabilités obtenues et puis (c) obtenir $K(s)$ à partir de $m(s)$ au moyen du théorème de codage de Levin-Chaitin.

Les sorties des machines sont groupées par fréquence (sont regroupées et comptées). La disposition des résultats, triés par fréquence, doit être analogue à la distribution de probabilité algorithmique, une distribution empirique de la semi-mesure de Levin $m(s)$.

Conclusions du chapitre

Les expériences suggèrent qu'on peut donner une définition stable de $m(s)$ et donc donner du sens à $K(s)$, en particulier pour les suites courtes. Nous suggérons que cette méthode, basée sur de tels dispositifs de classements, est une méthode efficace pour mesurer la complexité relative de suites courtes.

En réalisant ces expériences, impliquant plusieurs types de systèmes de calcul, on trouve des corrélations entre les classements de fréquence de suites, ce que nous interprétons comme une validation de la méthode proposée, qui doit cependant être confirmée par des calculs complémentaires.

1.1.2 Évaluation numérique d'une distribution expérimentale de Levin

Les définitions générales de K ne permettent pas de parler de la complexité de courtes suites (ou alors d'une manière tellement imprécise que cela n'a aucun intérêt). Divers chercheurs souhaiteraient utiliser K mais renoncent à le faire :

- soit parce qu'ils pensent que c'est impossible (incalculabilité, etc.)
- soit parce qu'ils croient que ça ne marche que pour des suites longues et que ce n'est pas cela dont ils ont besoin.

Court est relatif à la taille du compilateur (ou machine universelle choisie). On peut considérer courte toute suite de moins de 15 ou 20 chiffres.

De nombreux chercheurs utilisent d'autres mesures de "complexité", souvant mal justifiées, basées sur des évaluations combinatoires (nombre de facteurs) ou de nature statistique (par ex. l'entropie de Shannon). Notre méthode doit conduire à une définition stable de $K(s)$ susceptible d'un usage absolument général.

Pour calculer la complexité d'une suite quelconque, par exemple de 12 chiffres, nous produisons toutes les suites possibles de longueur 12 avec un système de calcul fixé. Cela nous donne une distribution de fréquences à partir de laquelle on classe la suite donnée et on calcule sa probabilité algorithmique, puis sa complexité algorithmique par le biais du théorème de codage de Levin-Chaitin.

Cette approche expérimentale impose aussi une limite, car évidemment même quand la procédure marchera pour des séquences plus longues, il y aura des restrictions imposées par les ressources de calcul de nos ordinateurs et le temps que nous serons disposés à passer pour obtenir une approximation.

Comparer si 0100110 est plus ou moins complexe que 110101100 au moyen d'algorithmes de compression ne marche pas bien. En effet, plus courte est une suite, plus important en proportion est le rôle de la taille de la machine universelle choisie (ou compilateur) pour la comprimer. En d'autres termes, la dépendance de la complexité algorithmique par rapport à la machine universelle choisie est si importante que la définition courante de la complexité algorithmique et son application pour les suites plus courtes n'a pas de sens. Notre méthode évite cette difficulté.

On définit $D(n)$ comme la fonction qui à tout s (suite finie de 0 et de 1) associe le quotient (nombre de fois qu'une machine de Turing de type $(n, 2)$ donne s) / nombre machines de Turing de type $(n, 2)$. La fonction $n \rightarrow D(n)$ est mathématiquement bien définie mais elle est non calculable (pour en savoir plus allez au chapitre 2), comme beaucoup d'autre fonctions en théorie de la calculabilité ; ce qui ne veut pas dire qu'on ne peut pas calculer $D(n)$ pour les petites valeurs de n.

Pour les petites valeurs de n, on peut savoir, pour toute machine, si elle s'arrête ou pas en utilisant la solution du problème du Castor affairé associé. C'est le cas si n =1, 2, 3, ou 4.

On calcule les distributions $D(n)$ et on produit une distribution de fréquence des suites binaires pour $n = 1, 2, 3$ et 4. À partir de la distribution on calcule $K(s) = -log_2(D(s))$ ce qui donne une évaluation de la complexité algorithmique de la suite s. On calcule donc $D(1)$, $D(2)$, $D(3)$ et $D(4)$. Pour $n > 4$ nous ne pourrons avoir pour l'instant que des approximations de $D(n)$.

La réalisation de ces calculs de $D(n)$ pour les petites valeurs de n est comparable au calcul des décimales de π au sens qu'on évalue des nombres fixés pour toujours. C'est aussi un calcul lié à l'évaluation du nombre Ω de Chaitin (voir [**?**]).

Pour $D(3)$, il a fallu déjà prendre en compte 15 059 072 machines. Pour $D(4)$, le nombre de machines est si grand qu'on ne peut pas les générer d'avance et les garder en mémoire, ni garder les résultats. Il faut calculer les classements en mémoire et mettre en place des compteurs assez sophistiqués.

Les méthodes testées pour $D(3)$ nous ont permis de calculer les 22 039 921 152 machines de Turing pour évaluer $D(4)$. Notons que la complexité de Kolmogorov pour suites courtes à partir de $D(n)$ n'est pas un nombre entier (comme lorsqu'on la définit comme la taille du plus court programme) mais un nombre réel. C'est un fait un avantage puisque cela permet d'avoir un classement plus fin avec moins d'ex aequo. La méthode qu'on propose donne une version expérimentale de K.

Conclusions du chapitre

Nous proposons donc un moyen de calculer une "distribution universelle" de Levin $D(n)$. On calcule $D(s)$ et donc $K(s)$ à partir de la formule $K(s) = -log_2(D(s))$. Les résultats trouvés sont conformes à notre intuition de ce qui est complexe et ce qui est simple. Nous esperons que les tables de complexité que nous publions seront des références définitives, utiles à ceux qui souhaitent disposer d'une "distribution empirique" bien fondée. Des calculs complémentaires encore plus massifs compléteront ces tables et étendront le champ d'applications possibles.

On étudie aussi les distributions générées par des automates cellulaires et on calcule la corrélation avec la distribution produite par des machines de Turing. Si plusieurs formalismes de calcul (automates cellulaires, machines de Turing, systèmes de Post, etc.) donnent le même ordre (ou à peu près), c'est sans doute qu'il y a une raison profonde à cela. Nos expériences suggèrent qu'un ordre universel pourrait exister et donc une sorte de complexité de Kolmogorov naturelle.

Ce travail ouvre la porte à de nouvelles applications de la complexité algorithmique, car on dispose d'une mesure approchable en pratique, par exemple pour faire de la compression de données.

1.2 Applications

1.2.1 Recherche des propriétés dynamiques des automates cellulaires et d'autres systèmes par des techniques de compression

Une méthode pour étudier les propriétés qualitatives dynamiques des machines à calculer, fondée sur le la comparaison de leur complexité algorithmique, en utilisant un algorithme de compression sans perte générale, est présentée. Il est montré que l'approche par la compression sur les automates cellulaires les classe en groupes selon leur comportement heuristique. Ces groupes montrent une correspondance avec les quatre principales classes de comportement identifiées par Wolfram[9]. Un codage de conditions initiales à base du code de Gray est aussi développé pour distinguer les conditions initiales, en augmentant leur complexité graduellement. Une méthode fondée sur une estimation d'un coefficient de détection des transitions de phase est aussi présentée. Ces coefficients permettent de mesurer la résistance ou la sensibilité d'un système à ses conditions initiales. Nous formulons une conjecture quant à la capacité d'un système à parvenir à l'universalité de calcul, liée aux valeurs de ce coefficient de transition de phase.

Conclusions du chapitre

On a pu clairement distinguer les différentes catégories de comportements étudiées par Wolfram. En calculant les longueurs comprimées des sorties des automates cellulaires, en utilisant un algorithme de compression général, nous avons trouvé qu'on distingue nettement deux groupes principaux et, regardant de plus près, deux autres groupes évidents entre les deux. Ce que nous avons trouvé semble soutenir le principe de Wolfram d'équivalence de calcul (ou PCE)[9].

On a également fourni un cadre de compression à base de transition de phase et une méthode pour calculer un exposant capable d'identifier et de mesurer l'importance d'autres propriétés dynamiques, telles que la sensibilité aux conditions initiales, la présence de structures dans l'espace ou la régularité dans le temps. Nous avons également formulé une conjecture en ce qui concerne le lien possible entre le coefficient de transition et la capacité d'un système à parvenir à l'universalité de calcul. Comme on peut le voir d'après les expériences présentées dans ce document, l'approche par compression et les outils qui ont été proposés sont très efficaces pour le classement et

la détection de plusieurs propriétés dynamiques des systèmes abstraits. En outre, la méthode ne dépend pas du système étudié et peut s'appliquer à n'importe quel système de calcul abstrait ou à des données provenant d'une source quelconque. Il peut également être utilisé pour calculer les distributions a priori et faire des prévisions concernant l'évolution future d'un système.

Ces idées constituent un cadre à base de compression pour enquêter sur les propriétés dynamiques des automates cellulaires et d'autres systèmes, dont nous sommes certains qu'il aura d'autres applications.

1.2.2 Classification d'images par complexité organisée : une application de la profondeur logique de Bennett

Nombreuses sont les applications qui ont utilisé la complexité algorithmique pour classer des objets en utilisant des algorithmes de compression de données (sans perte) :

- Arbres phylogénétiques à partir de séquences génétiques (Varré et Delahaye[3]) ;
- Arbres représentant les parentés entre les langues indoeuropéennes en partant des traductions de *La Déclaration universelle des droits de l'homme* (Cilibrasi et Vitányi[2]) ;
- Comparaisons de textes littéraires (Cilibrasi et Vitányi[**?**]) ;
- Repérage de la fraude et du plagiat [1] ;
- Classification de morceaux de musique (Cilibrasi, Vitányi et de Wolf[**?**]) ;
- Détection du "spam" (Richard et Doncescu[7]).

Une idée nouvelle est proposée ici fondée sur le concept de profondeur logique[**?**] développé par Charles Bennett : évaluer et classer des images par l'évaluation du temps de décompression des versions comprimées (sans perte) des images.

La taille du fichier compressé est interprétée comme une valeur approchée de la complexité des données contenues dans le fichier brut. Le temps nécessaire à la décompression du fichier est donc naturellement interprété comme une valeur approchée de la profondeur logique de la donnée contenue dans le fichier brut.

Contrairement à l'application de la notion de complexité algorithmique par elle seule, l'ajout de la notion de profondeur logique a pour résultat une

prise en compte de la richesse de leur organisation structurale ou complexité organisée.

Sur une machine parfaite idéale qui ne ferait que de la décompression, on pourrait effectuer cette mesure une seule fois. La répéter N fois donnerait N fois le même résultat. Par contre, on lance un programme qui lance des programmes par l'intermédiaire d'un système d'exploitation, dans un environnement contraint (par exemple Mac OS X). Ce programme est chargé d'ouvrir un fichier, de décompresser son contenu et de l'enregistrer dans un fichier de destination. Or, sur plusieurs essais, on ne mesure pas nécessairement toujours la même valeur (temps d'exécution) et on est obligé de répéter plusieurs fois cette mesure pour obtenir une moyenne.

Il a fallu donc décompresser chaque image plusieurs fois de suite et on s'est arrangé pour que la plupart des programmes de l'ordinateur sur lequel on fait les mesures soient arrêtés, par exemple les programmes du réseaux, les programmes en mémoire, etc.

Du fait de ces phénomènes d'instabilité toujours présents dans les ordinateurs modernes, quand on le fait cinq fois de suite par exemple pour cinq images A, B, C, D et E, ça ne revient sans doute pas au même de considérer les ordres :

- AAAAABBBBBCCCCCDDDDDEEEEE ou
- ABCDEABCDEABCDEABCDEABCDE (toujours le même ordre) ou
- ABCDEDEBCADBACEDABCEECDBA (en changeant l'ordre à chaque fois)

L'expérimentation indique que la dernière méthode est la meilleure, car le contexte de l'ordinateur change et qu'on a plus de chance que chaque image se retrouve dans une série de contextes variés, et dans l'ensemble à peu près identiques pour chaque image. La méthode des temps de décompression sur une machine moderne est délicate à mettre en œuvre, mais il est certain que, pour qu'elle marche un peu et qu'elle soit crédible, il fallait réussir à la stabiliser comme on l'a fait.

Il se peut qu'en mesurant chaque temps de décompression pour la même image on obtienne des temps de calcul bien stables, sauf dans certains cas. Nous étudions dans ce chapitre ces techniques et nous donnons une réponse à plusieurs questions méthodologiques, en particulier quant aux différences en tant que mesures de complexité, entre la complexité algorithmique K et la profondeur logique P.

Conclusions du chapitre

Nous présentons une méthode pour estimer la complexité d'une image basée sur le concept de profondeur logique.

On peut voir que :

- De K à P, les images d'êtres vivants, de microprocesseurs et de villes avancent, ce qui est naturel car ce sont les objets possédant une complexité d'organisation maximale.
- De K à P, les mécanismes et les produits manufacturés par l'homme, comme une montre ou l'écriture avancent, ce qui encore est le résultat attendu.
- De K à P, les images qui semblent les plus aléatoires reculent ainsi que les plus simples : les courbes de Peano et les courbes symétriques. Une fois encore, c'est ce que l'intuition faisait espérer.

Le fait que ces images "aléatoires" se retrouvent avec les images les plus simples (cristal) est absolument parfait.

Les images de courbes de Peano sont considérées comme plus organisées. Cela est assez satisfaisant : leur organisation n'est pas simple (ce n'est pas une organisation répétitive). Les êtres vivants et les microprocesseurs sont les mieux placés, ce qui est très satisfaisant, car ils sont certainement parmi les objets les plus complexes. Les images inversées se placent toujours à côté des images normales dans le cas de K et pas trop loin l'une de l'autre pour P, ce qui est très bien.

Notons encore que le fait que les objets ayant un K très élevé (objects aléatoires) se retrouvent avec les objets les plus simples (P très faible) dans le classement donné par P est ce que nous attendions.

Il y a sans doute bien d'autres choses à dire mais tous les changements de K à P confirment l'idée théorique de Bennett à la base de notre travail, et il montre donc que le concept de profondeur logique est efficace. En résumé :

1. Le concept de profondeur logique peut être mis en œuvre par une méthode faisable pour traiter de données réelles.
2. Des algorithmes de compression et une méthode spécifique ont été décrits dans ce travail, et on a montré qu'ils fonctionnaient et étaient utiles pour identifier et classer des images en fonction de leur complexité d'organisation.
3. La procédure décrite ici constitue une méthode non supervisée d'évaluation du contenu informationnel d'une image par la complexité d'organisation.

1.2.3 Le fossé de Sloane

L'encyclopédie des suites numériques (couramment abrégé OEIS) créée par Neil J.A. Sloane (http ://www.research.att.com/∼njas/sequences/) est une base de données de suites d'entiers. Elle est une référence dans le domaine des suites d'entiers pour les mathématiciens professionnels et amateurs. Elle recensait 158000 suites en mai 2009.

Nous nous intéresserons à la distribution du nombre d'occurrences $N(n)$ d'un entier n dans l'encyclopédie de Sloane. Considérons le nombre de propriétés d'un entier, $N(n)$, en le mesurant par le nombre de fois où n apparaît dans le fichier numérique de l'encyclopédie de Sloane. La valeur $N(n)$ mesure l'intérêt de n.

On obtient un nuage à l'allure régulière dans la représentation en échelle logarithmique de $N(n)$ en fonction de n. Le nuage est divisé en deux parties séparées par une zone claire, comme si les nombres se séparaient naturellement en deux catégories. Ce *fossé* est une zone creuse inattendue repérée par Philippe Guglielmetti [1].

Notre but est de décrire la forme du nuage, puis de formuler une hypothèse explicative de cette forme. L'existence du fossé de Sloane est-elle naturelle, ou demande-t-elle une explication spécifique ?

Conclusions du chapitre

Nous montrons que la complexité algorithmique est un outil qui donne une indication sur l'allure que devrait présenter la courbe représentative de N fondée sur une mesure d'importance "objective". Si la forme générale du nuage est prévisible, la présence du fossé de Sloane a, en revanche, bien plus interpellé les observateurs. Ce fossé n'a pas, à notre connaissance, pu être expliqué par des considérations uniquement numériques et indépendantes de la nature humaine du travail mathématique. La complexité algorithmique laisse en effet prévoir une certaine "continuité" de N, puisque la complexité de $n + 1$ est toujours proche de celle de n. La discontinuité qui prend corps dans le fossé de Sloane est donc difficilement attribuable à des propriétés purement mathématiques indépendantes des contingences sociales. Comme nous le montrons, elle s'explique très bien par le fonctionnement de la recherche, qui entraîne la surreprésentation de certains nombres de faible ou moyenne complexité. Ainsi, le nuage de points représentant la fonction N

1. Sur son site http ://drgoulu.com/2009/04/18/nombres-mineralises/
Consulté le 3 août 2009.

présente-t-il simultanément des caractéristiques que l'on peut comprendre comme de nature mathématiques et humaines.

1.3 Réflexions

1.3.1 Sur la nature algorithmique du monde

Ce chapitre est consacré à une réflexion sur les connexions entre la théorie et le monde physique. Nous prenons des sources d'information dans le monde réel et les comparons avec ce qu'un monde algorithmique aurait produit. Si le monde était composé uniquement à partir de règles déterministes, alors le monde serait "algorithmique".

Notre idée est que ce type d'hypothèses est testable par le biais de la distribution universelle de Levin (ailleurs elle est appelée semi mesure de Solomonoff-Levin). Notre travail a consisté à mener de tels tests qui, bien sûr, ne peuvent pas être faciles et seront sujets à discussion, voire à contestation. Il semble important qu'un dilemme de nature fondamentale (philosophique) puisse donner lieu à ce que les physiciens appellent une expérience cruciale. Comme bien sûr rien n'est parfaitement simple, nous ne pouvons pas être absolument certain que nous détenons les clefs d'une expérience cruciale pour identifier la nature profonde des mécanismes du monde physique (algorithmiques ou non), mais notre piste est sérieuse, car basée sur des conséquences de la théorie de l'information algorithmique, et en particulier sur la notion de probabilité algorithmique.

Si on accepte que le monde suive des règles, jusqu'à quel point le monde ressemble-t-il à un monde purement algorithmique ? Pour tenter de s'approcher d'une formulation sérieuse de cette question, nous avons envisagé des tests statistiques qui mettent en évidence une ressemblance, d'abord entre les mondes algorithmiques possibles (un par modèle de calcul), puis par rapport aux sources de données empiriques.

En d'autres termes, si on accepte que :

1. le monde est une sorte de grande machine à calculer (ou une famille de machines à calculer, ou un réseau d'automates mais toujours Turing-calculables) et que
2. les programmes des machines composant le monde sont choisis au hasard

alors $m(s)$ pourrait être la probabilité qu'en observant le monde au hasard on tombe sur s.

Néanmoins, la nature de la distribution des sorties à partir de la fréquence dans laquelle les suites sont produites fournit des informations sur le processus originel. Une distribution qui s'approche de ce qu'on attend par la mesure de Levin suggère que le processus générateur mène des calculs qui ressemblent à ceux qui pourraient être menés par des programmes sur une machine universelle de Turing (formalisme sur lequel la théorie de la complexité est fondée et définie).

C'est la fréquence d'apparition de s, et donc la probabilité $m(s)$, qui détermine si l'origine des suites est générée par un programme de calcul à l'origine, de la même façon que trouver une distribution gaussienne suggérerait fortement que le processus d'origine a été le résultat d'un processus aléatoire au sens classique.

Toute l'expérimentation repose sur deux types de systèmes : d'une part des automates abstraits, comme les machines de Turing, et les automates cellulaires, qui sont eux-mêmes comparés l'un à l'autre (pour tester s'ils arrivent, à peu près, à une même distribution), et d'autre part des sources de données réelles contenant des informations venant du monde physique : des images en noir et blanc, des séquences d'ADN et de données dans un disque dur quelconque.

Dans tous les cas, on va représenter la taille de la suite à rechercher et à comparer, à chaque étape. D'habitude, une suite est représentée par un $s \in \{0,1\}^*$. Dans le cas des automates abstraits, nous observons leur sortie (suites binaires) après les avoir laissés tourner pendant un certain nombre d'étapes. Dans le cas des données réelles, nous transformons leur contenu en suites binaires. Et dans les deux cas, ces suites sont partitionnées selon la longueur donnée, puis regroupées par fréquences. De tels regroupements donnent une distribution expérimentale de Levin.

Conclusions du chapitre

Nous ne pouvons pas apporter une réponse définitive aux questions que nous posons, mais notre méthode indique une piste pour explorer ce type de questions, et cela est nouveau. Avant ces considérations sur le concept de probabilité algorithmique, personne n'envisageait de méthode pour départager les différentes hypothèses (ou formuler plus précisément des hypothèses mixtes que nous n'excluons pas).

Une hypothèse opposée à la nôtre serait celle d'un monde déterminé uniquement par le hasard. L'étude d'un tel monde montrerait une distribution uniforme ou en tout cas très différente de celle que la mesure de Solomonoff-Levin laisse attendre.

Nos résultats montrent qu'il existe une corrélation, en termes de fréquences de séquences, dans les données en sortie pour plusieurs systèmes, dont des systèmes abstraits comme les automates cellulaires et les machines de Turing, ainsi que des prélèvements d'information du monde réel, tel que les images et les fragments d'ADN humain. Nos résultats suggèrent donc que, derrière tous ces systèmes, il pourrait avoir une distribution partagée en accord avec ce que prédit la probabilité algorithmique.

1.3.2 Une approche algorithmique du comportement des marchés financiers

Cette étude essaie d'appliquer aux recherches sur les variations des marchés financiers des idées issues de la théorie algorithmique de l'information, notamment des travaux de Levin, pour expliquer l'écart connu des données du marché par rapport à la distribution théorique généralement. Cela signifie donc que les variables ne sont pas indépendantes et donc pas complètement aléatoires. Notre analyse va donc dans le sens des critiques adressées par Benoit Mandelbrot[6] aux instruments mathématiques traditionnellement utilisés dans ce domaine. Cependant, l'approche est complètement nouvelle et différente de celle de Mandelbrot.

Pour étudier le marché par le biais des suites binaires, on note :

- 1 pour une hausse et
- 0 pour une baisse.

En utilisant les distributions de fréquences de séries quotidiennes de prix de clôture sur plusieurs marchés boursiers, nous cherchons à savoir si l'écart prévisible par la probabilité algorithmique (une distribution de loi de puissance) peut rendre compte des déviations attendues par la théorie aléatoire des marchés (une distribution log-normale). Notre étude constitue un point de départ pour de plus amples recherches sur les marchés en tant que systèmes soumis à des règles, dotés d'une composante algorithmique, malgré leur apparente nature aléatoire. L'utilisation de la théorie de la complexité algorithmique fournit un ensemble de nouveaux outils d'investigation, pouvant s'appliquer à l'étude du phénomène d'attribution de prix sur les marchés financiers.

En économie, la nature de la dynamique à laquelle sont soumises les données est différente. Les structures sont rapidement effacées par l'activité économique elle-même, au cours de la recherche d'un équilibre économique (par exemple dans la Bourse).

En posant l'hypothèse algorithmique qu'il existe une composante basée sur une règle, contrairement à un processus stochastique, dans les marchés, on peut appliquer les outils de la théorie de l'information algorithmique, de la même façon que la supposition de distributions aléatoires conduit à l'application de la machinerie traditionnelle de la théorie classique des probabilités.

Si cette hypothèse se révélait vraie, la théorie dit que la distribution de Solomonoff-Levin est l'indicateur optimal. Autrement dit, on pourrait lancer un grand nombre de machines pour simuler le marché, et m , la probabilité algorithmique basée sur la distribution universelle de Levin, fournirait certaines indications sur la direction et la taille particulière d'un prix, basées sur le fait que le marché possède un élément obéissant à une famille de règles.

Nous pensons que l'étude des distributions de fréquences et l'application de probabilités algorithmiques pourraient constituer un outil pour estimer et finir par comprendre le processus d'assimilation de l'information dans les marchés, rendant possible la caractérisation du contenu informationnel des prix.

Conclusions du chapitre

Les suites des changements des prix sont classées en fonction du nombre de 1. C'est pour cela que la suite 0000 est en dernier. La raison en est sans doute que les 1 et les 0 ne sont pas équidistribués. L'explication se trouve dans l'asymétrie des marchés (les courbes n'ont pas la même allure quand on inverse l'axe du temps) : ils montent plus lentement qu'ils ne descendent (mais quand ils montent, ils montent moins que lorsqu'ils descendent et donc ça se compense), ainsi il y a plus de 1 que de 0. A priori ici, cela n'a rien à voir avec la complexité algorithmique.

La spéculation sur les cours boursiers a pour effet de supprimer les structures exploitables. Le déséquilibre en 1 et 0 n'est pas exploitable s'il est compensé par des hausses plus faibles que les baisses. Il a donc fallu *effacer* cette asymétrie.

Pour savoir si la complexité algorithmique a quelque chose à dire, il fallait comparer un assez grand nombre de suites ayant le même nombre de 1 et de même longueur : par exemple toutes les suites de longueur 10 et ayant cinq 1. On verrait peut-être alors que les suites les plus simples (0000011111,

0101010101, etc.) sont placées devant les plus complexes (01001110110). Ce n'était pas du tout certain, mais ça a été le cas.

Les corrélations trouvées dans les expériences décrites suggèrent que la distribution de Levin pourrait bien être un moyen de calculer et d'évaluer ce segment de cet aspect algorithmique du marché.

1.3.3 Complexité algorithmique versus complexité de temps de calcul : recherche dans l'espace des petites machines de Turing

Parmi les diverses mesures de complexité, il y a des mesures axées sur la description minimale d'un programme et d'autres sur la quantification des ressources (espace, temps, énergie) utilisées par un calcul.

Le but est de trouver des compromis entre la complexité algorithmique et la complexité de temps de calcul par une exploration exhaustive et systématique des fonctions calculées par l'ensemble des machines de Turing à 2 symboles avec 2, 3 et 4 états, en portant l'attention sur l'espace occupé sur les rubans et les temps d'exécution des machines correspondant aux fonctions calculées par ces machines lorsqu'elles ont accès à plus d'états.

Le problème de l'arrêt et d'autres contraintes importantes pour cette approche expérimentale, y compris le problème d'extensionnalité (décider si deux machines calculent la même fonction), ont été surmontés en prenant une allure pragmatique. Par exemple, on considère que deux fonctions sont la même fonction si pour 21 valeurs d'entrée écrites sur le ruban des deux machines, on a sur les rubans le même contenu au moment de l'arrêt, et on considère que deux machines calculent le même algorithme si les deux machines calculent la même fonction et si elles le font en occupant le même temps et espace du ruban.

Conclusions du chapitre

On a obtenu des évaluations exactes en ce qui concerne les temps de calcul, dévoilant des proprietés du microcosme des petites machines de Turing et fournissant des chiffres sur les fonctions qu'elles calculent. L'analyse des données nous a fourni des fonctions dont les classes de temps de calcul ont été comparées parmi les différents espaces des machines à différents nombres d'états. Au début, l'intuition nous disait qu'une machine de Turing qui calcule la même fonction avec plus d'états pourrait calculer la fonction en moins

de temps. Par contre, on a trouvé que plus une machine a accés à plus d'états, plus il est probable que la machine gaspille ces nouvelles ressources disponibles. Ce phénomène a été trouvé à chaque fois qu'on passait de 2 à 3 états et de 3 à 4 états.

On a trouvé des cas où une fonction était parfois calculée plus vite sans franchir une classe de complexité de temps. Des machines qui arrivaient à calculer des fonctions, par exemple, deux fois plus vite mais dans ce que nous considérions la même classe de temps (par ex. $O(n^2)$ et $O(2n^2)$ appartenant toutes les deux à la classe de temps quadratique). Jamais aucune fonction n'a été calculée plus vite qu'une machine qui calculait la même fonction avec moins d'états ; plus vite signifierait, par exemple, une machine qui calcule une fonction en temps $O(n)$ qui est calculée en temps $O(n^2)$ par une machine à moins d'états.

D'autres phénomènes intéressants tels que certaines transitions de phase dans les courbes des temps d'arrêt des machines on été expliqués par une étude sur la manière dont on encode les entrées sur les rubans et le fonctionnement lui-même des machines. D'autres phénomènes et des machines curieuses ont été aussi découverts, tel est le cas des machines qui calculent la fonction identité dans un temps exponentiel, ce qui exemplifie l'une de nos découvertes : une présence croissante de machines de plus en plus lentes qui ne profitent pas des nouvelles ressources. La courbe de complexité algorithmique (nombre d'états) contre complexité de temps, confirme ce que nous avons trouvé, notamment un décalage systématique des temps d'exécution des machines qui calculent la même fonction, c'est-à-dire que la moyenne de temps de calcul des machines devient de plus en plus longue, indiquant que les machines tendent à occuper toutes les ressources à portée de main.

1.4 Résumé des principales contributions

Les principales contributions de cette thèse sont les suivantes :

- On a construit des distributions de fréquences à partir de plusieurs modèles de calcul. Les classements de séquences dans ces distributions expérimentales sont relativement stables et corrélés, ce qui suggère une distribution naturelle, au moins sous certaines conditions (petite taille des séquences). On a conclu qu'une distribution naturelle pourrait nous permettre de réduire l'impact de la constante additive à laquelle la mesure de complexité algorithmique est sujette.

– On a construit des distributions de fréquences à partir de plusieurs sources physiques du monde réel et on les a comparées avec celles obtenues à partir des modèles de calcul algorithmique. Les classements de séquences dans ces distributions expérimentales ont été trouvés stables, et là encore, corrélées. Le but a été de concevoir un test statistique valide pour tester l'assertion que le monde est de nature algorithmique, plutôt qu'aléatoire.
– On a réussi à classer de manière automatique, grâce à l'utilisation des méthodes de compression et donc de l'approximation de leur complexité algorithmique, plusieurs évolutions de systèmes abstraits de calcul, en particulier des automates cellulaires. On a prouvé que cette classification coïncide avec la classification des quatre comportements identifiés par Wolfram. On a défini un coefficient de transition qui mesure la stabilité des systèmes aux conditions initiales et qui permet de mesurer l'homogénéité d'un système par rapport au changement des entrées et donc, d'une certaine façon, sa manière de transmettre l'information. On en a tiré une conjecture intéressante qui fait un rapport entre la magnitude du coefficient défini et la capacité d'un système de calcul à se comporter de manière universelle, c'est-à-dire simuler n'importe quel autre système.
– On a trouvé et mis en place une application concrète de la profondeur logique de Bennett pour classer des images par leur complexité organisée. La méthode a réussi à classer les objets conformément à l'intuition, en donnant les bases techniques pour appliquer les méthodes dans d'autres contextes. On présente aussi les différences pratiques entre la mesure de la complexité algorithmique pleine et la profondeur logique, ce qui illustre leurs différents atouts. On envisage plusieurs applications dans plusieurs domaines, en particulier la biologie, pour détecter automatiquement la complexité d'un objet à travers une image.
– On a étudié le rapport et les échanges entre différents types de ressources, en particulier ceux liés à la taille d'un programme (donc sa complexité algorithmique) et le temps de calcul pris pour calculer certaines fonctions mathématiques. On a trouvé des comportements intéressants et on a étudié tout l'espace des petites machines de Turing, ainsi que toutes les fonctions (et les temps de calcul) qu'elles calculent.
– On a exploré et ouvert un chemin pour chercher si l'écart par rapport à la distribution d'une série équiprobable, prévisible par la probabilité algorithmique, peut rendre compte des déviations des marchés financiers par rapport à une distribution log-normale, connue pour ne pas être suivie par les données des marchés. L'étude des distributions de fréquences et l'application de probabilités algorithmiques peuvent

constituer un outil pour estimer et aider à comprendre le processus d'assimilation de l'information dans les marchés, rendant possible la caractérisation du contenu informationnel des prix.
– On a calculé et fournit des tables de fréquences de suites binaires qui donnent des évaluations numériques de la complexité algorithmique (et de la profondeur logique de Bennett) en utilisant le problème du Castor affairé pour résoudre le problème de l'arrêt pour des machines de Turing à 4 états. À travers m, la semi-mesure de Levin aussi appelée distribution universelle, on a calculé numériquement la complexité algorithmique de suites assez courtes.
– On a appliqué la théorie algorithmique de l'information pour montrer que la décroissance de la courbe de la base de données d'entiers de Sloane est le résultat d'un phénomène dû à la complexité algorithmique, et on explique le nuage par un phénomène social qui favorise certaines suites.

Les chapitres de cette thèse ont été publiés ou ont été soumis à des revues à comités de lecture.

Voici la liste par ordre chronologique :

– J.-P. Delahaye et H. Zenil, "On the Kolmogorov-Chaitin complexity for short sequences', dans *Randomness and Complexity : From Leibniz to Chaitin,* edité par Cristian S. Calude (University of Auckland, New Zealand), World Scientific, 2007.
– H. Zenil, "Compression-based investigation of the dynamical properties of cellular automata and other systems," publié par le journal of Complex Systems, 14 : 2, 2010.
– J. Joosten, F. Soler et H. Zenil, "Program-size versus Time Complexity, Slowdown and Speed-up phenomena in Small Turing Machines", publié dans proceedings 3rd. International workshop on Physics and Computation 2010, International Journal of Unconventional Computing, 2011.
– F. Soler-Toscano, J. Joosten et H. Zenil, "Complejidad descriptiva y computacional en máquinas de Turing pequeñas" (traduction espagnole de l'article précédent), Actas de las V Jornadas Ibéricas, Logica Universal e Unidade da Ciência, CFCUL, 2010.
– H. Zenil, J.-P. Delahaye et C. Gaucherel, "Information content characterization and classification of images by physical complexity," soumis à Complexity, 2011.
– H. Zenil et J-P. Delahaye, On the Algorithmic Nature of the World', in Mark Burgin, Gordana Dodig-Crnkovic (eds), "Information and Com-

putation," World Scientific, 2011.
- H. Zenil et J.-P. Delahaye, "An algorithmic information-theoretic approach to the behaviour of financial markets," themed issue on Nonlinearity, Complexity and Randomness, publié par Journal of Economic Surveys, 2011.
- J-P. Delahaye et H. Zenil, "Numerical Evaluation of Algorithmic Complexity for Short Strings : A Glance into the Innermost Structure of Randomness", soumis à Applied Mathematics and Computation .
- J.-P. Delahaye, N. Gauvrit et H. Zenil, "Sloane's Gap : Do Mathematical and Social Factors Explain the Distribution of Numbers in the OEIS ?" soumis au Journal of Integer Sequences. Version franąise publiée sous le titre "Le fossé de Sloane" publiée par Mathématiques et Sciences Humaines - Mathematics and Social Sciences, 2011.

On a aussi présenté ce travail dans plusieurs congrès et ateliers.

Voici la liste du plus récent au plus ancien :

- *Workshop Physics and Computation 2010*. Luxor, Égypte, Septembre 2010.
- *Workshop on Nonlinearity, Complexity and Randomness 2009*, Département d'économie de l'Université de Trento, Italie. Décembre 2009.
- *Séminaire ECCO*, Université Libre de Belgique. Bruxelles, Belgique, Novembre 2009.
- Amphithéâtre Kurt Gödel, Université des Sciences et Technologies de Lille, LIFL, France, Septembre 2008.
- Grande Salle, *Séminaire Philmat*, Institut d'Histoire et de Philosophie des Sciences, IHPST, Paris 1, Paris, France. Décembre 2008.
- *Summer School on Randomness*, Math department, University of Florida, USA. May 2008.
- *NKS Science Conference*, University of Vermont, Burlington, USA. July 2007.

Avertissement

Le fil directeur des chapitres de cette thèse de recherche est une question fondatrice et pragmatique. Il s'agit d'une approche expérimentale de la théorie algorithmique de la complexité, d'un côté pour approcher des valeurs numériques, et ensuite donner des évaluations exactes pour un formalisme

fixé. D'un autre côté, il s'agit d'une réflexion par rapport aux connexions entre le monde physique (en particulier les processus et phénomènes naturels et sociaux dont nous essayons de donner une explication théorique) et la théorie algorithmique de l'information.

J'espère, à travers ces quelques pages, avoir donné envie au lecteur de poursuivre la lecture du reste de la thèse, composée de chapitres indépendants.

Bibliographie

[1] C. Basile, D. Benedetto, E. Caglioti, and M. Degli Esposti, An example of mathematical authorship attribution. *Journal of Mathematical Physics,* 49 :125211–125230, 2008.

[2] R. Cilibrasi et P.M.B. Vitányi. The Google Similarity Distance, *IEEE Trans. Knowledge and Data Engineering,* 19 :3, 370-383, 2007. [**?**] R. Cilibrasi, P.M.B. Vitányi, and R. de Wolf. Algorithmic Clustering of Music Based on String Compression, *Computer Music Journal*, 28 :4, pp. 49–67, 2004.

[3] J.-S. Varré et J.-P. Delahaye, Transformation distances : a family of dissimilarity measures based on movements of segments. *Bioinformatics/computer Applications in The Biosciences - Bioinformatics*, vol. 15, no. 3, pp. 194-202, 1999.

[4] M. Li, X. Chen, X. Li, B. Ma et P.M.B. Vitányi, *Similarity Distance and Phylogeny,* Manuscript, 2002.

[5] M. Li, X. Chen, X. Li, B. Ma et P.M.B. Vitányi, The Similarity Metric, *IEEE Transactions on Information Theory*, 2003.

[6] B. Mandelbrot et R.L. Hudson, *The (Mis)Behavior of Markets : A Fractal View of Risk, Ruin, and Reward,* Basic Books, 2004.

[7] G, Richard, A. Doncescu, Spam filtering using Kolmogorov complexity analysis. *IJWGS* 4(1) : 136-148 (2008)

[8] T. Rado, On noncomputable Functions, *Bell System Technical J.* 41, 877–884, May 1962.

[9] S. Wolfram, *A New Kind of Science.* Wolfram Media, 2002.

[10] H. Zenil, J.-P. Delahaye, C. Gaucherel, Image Characterization and Classification by Physical Complexity, arXiv :1006.0051v3 [cs.CC], 2010.

Deuxième partie

Fondements

Chapitre 2

On the Kolmogorov-Chaitin Complexity for Short Sequences

Long abstract (with the very first ideas of this thesis) published in J-P. Delahaye and H. Zenil, "On the Kolmogorov-Chaitin Complexity for Short Sequences", in C.S. Calude (ed.), *Randomness and Complexity: From Leibniz to Chaitin*, World Scientific, 2007.

Among the several new ideas and contributions made by Gregory Chaitin to mathematics is his strong belief that mathematicians should transcend the millenary theorem-proof paradigm in favor of a quasi-empirical method based on current and unprecedented access to computational resources[3]. In accordance with that dictum, we present in this paper an experimental approach for defining and measuring the Kolmogorov-Chaitin complexity, a problem which is known to be quite challenging for short sequences–shorter for example than typical compiler lengths.

The Kolmogorov-Chaitin complexity (or algorithmic complexity) of a string s is defined as the length of its shortest description p on a universal Turing machine U, formally $K(s) = min\{lp| : U(p) = s\}$. The major drawback of K, as measure, is its uncomputability. So in practical applications it must always be approximated by compression algorithms. A string

is uncompressible if its shorter description is the original string itself. If a string is uncompressible it is said that the string is random since no patterns were found. Among the 2^n different strings of length n, it is easy to deduce by a combinatoric argument that one of them will be completely random simply because there will be no enough shorter strings so most of them will have a maximal K complexity. Therefore many of them will remain equal or very close to their original size after the compression. Most of them will be therefore random. An important property of K is that it is nearly independent of the choice of U. However, when the strings are short in length, the dependence of K on a particular universal Turing machine U is higher producing arbitrary results. In this paper we will suggest an empirical approach to overcome this difficulty and to obtain a stable definition of the K complexity for short sequences.

Using Turing's model of universal computation, Ray Solomonoff[9, 10] and Leonid Levin[7] developed a theory about a universal prior distribution deeply related to the K complexity. This work was later known under several titles: universal distribution, algorithmic probability, universal inference, among others[6, 5]. This algorithmic probability is the probability $m(s)$ that a universal Turing machine U produces the string s when provided with an arbitrary input tape. $m(s)$ can be used as a universal sequence predictor that outperforms (in a certain sense) all other predictors[5]. It is easy to see that this distribution is strongly related to the K complexity and that once $m(s)$ is determined so is $K(s)$ since the formula $m(s)$ can be written in terms of K as follows $m(s) \approx 1/2^{K(s)}$. The distribution of $m(s)$ predicts that non-random looking strings will appear much more often as the result of a uniform random process, which in our experiment is equivalent to running all possible Turing machines and cellular automata of certain small classes according to an acceptable enumeration. By these means, we claim that it might be possible to overcome the problem of defining and measuring the K complexity of short sequences. Our proposal consists of measuring the K complexity by reconstructing it from scratch basically approximating the algorithmic probability of strings to approximate the K complexity. Particular simple strings are produced with higher probability (i.e. more often produced by the process we will describe below) than particular complex strings, so they have lower complexity.

Our experiment proceeded as follows: We took the Turing machine (TM) and cellular automata enumerations defined by Stephen Wolfram[11]. We let run (a) all 2-state, 2-symbol Turing machines, and (b) a statistical sample of the 3-state, 2-symbol ones, both henceforth denoted as $TM(2,2)$ and $TM(3,2)$.

Then we examine the frequency distribution of these machines' outputs performing experiments modifying several parameters: the number of steps, the length of strings, pseudo-random vs. regular inputs, and the sampling sizes.

For (a) it turns out that there are 4096 different Turing machines according to the formula $(2sk)^{sk}$ derived from the traditional $5-tuplet$ description of a Turing machine: $d(s_{\{1,2\}}, k_{\{1,2\}}) \rightarrow (s_{\{1,2\}}, k_{\{1,2\}}, \{1, -1\})$ where $s_{\{1,2\}}$ are the two possible states, $k_{\{1,2\}}$ are the two possible symbols and the last entry {1,-1} denotes the movement of the head either to the right or to the left. From the same formula it follows that for (b) there are 2985984 so we proceeded by statistical methods taking representative samples of size 5000, 10000, 20000 and 100000 Turing machines uniformly distributed over $TM(3, 2)$. We then let them run 30, 100 and 500 steps each and we proceeded to feed each one with (1) a (pseudo) random (one per TM) input and (2) with a regular input.

We proceeded in the same fashion for all one dimensional binary cellular automata (CA), those (1) which their rule depends only on the left and right neighbors and those considering two left and one right neighbor, henceforth denoted by $CA(t, c)$ [1] where t and c are the neighbor cells in question, to the left and to the right respectively. These CA were fed with a single 1 surrounded by 0s. There are 256 one dimensional nearest-neighbor cellular automata or $CA(1, 1)$, also called Elementary Cellular Automata[11]) and 65536 $CA(2, 1)$.

To determine the output of the Turing machines we look at the string consisting of all parts of the tape reached by the head. We then partition the output in substrings of length k. For instance, if $k = 3$ and the Turing machine head reached positions 1, 2, 3, 4 and 5 and the tape contains the symbols {0,0,0,1,1} then we increment the counter of the substrings 000, 001, 011 by one each one. Similar for CA using the "light cone" of all positions reachable from the initial 1 in the time run. Then we perform the above for (1) each different TM and (2) each different CA, giving two distributions over strings of a given length k.

We then looked at the frequency distribution of the outputs of both classes TM and CA [2], (including ECA) performing experiments modifying several

1. A better notation is the $3-tuplet\ CA(t, c, j)$ with j indicating the number of symbols, but because we are only considering $2-symbol$ cellular automata we can take it for granted and avoid that complication.

2. Both enumeration schemes are implemented in Mathematica calling the functions CelullarAutomaton and TuringMachine, the latter implemented in *Mathematica* ver. 6.0

parameters: the number of steps, the length of strings, (pseudo) random vs. regular inputs, and the sampling sizes.

An important result is that the frequency distribution was very stable under the several variations described above allowing to define a *natural* distribution $m(s)$ particularly for the top it. We claim that the bottom of the distribution, and therefore all of it, will tend to stabilize by taking bigger samples. By analyzing the following diagram it can be deduced that the output frequency distribution of each of the independent systems of computation (TM and CA) follow an output frequency distribution. We conjecture that these systems of computation and others of equivalent computational power converge toward a single distribution when bigger samples are taken by allowing a greater number of steps and/or bigger classes containing more and increasingly sophisticated computational devices. Such distributions should then match the value of $m(s)$ and therefore $K(s)$ by means of the convergence of what we call their experimental counterparts $m_e(s)$ and $K_e(s)$. If our method succeeds as we claim it could be possible to give a stable definition of the K complexity for short sequences independent of any constant.

By instance, the strings 0101 and 1010 were grouped in second place, therefore they are the second most complex group after the group composed by the strings of a sequence of zeros or ones but before all the other 2^n strings. And that is what one would expect since it has a very low K complexity as prefix of a highly compressible string 0101 . . .. In favor of our claims about the nature of these distributions as following $m(s)$ and then approaching $K(s)$, notice that all strings were correctly grouped with their equivalent category of complexity under the three possible operations/symmetries preserving their K complexity, namely reversion (sy), complementation (co) and composition of the two ($syco$). This also supports our claim that our procedure is working correctly since it groups all strings by their complexity class. The fact that the method groups all the strings by their complexity category allowed us to apply a well-known lemma used in group theory to enumerate actual different cases, which let us present a single representative string for each complexity category. So instead of presenting a distribution with 1024 strings of length 10 it allows us to compress it to 272 strings.

We have also found that the frequency distribution from several real-world data sources also approximates the same distribution, suggesting that they probably come from the same kind of computation, supporting contemporary claims about nature as performing computations[11, 8]. The paper available online contains more detailed results for strings of length $k = 4, 5, 6, 10$ as well

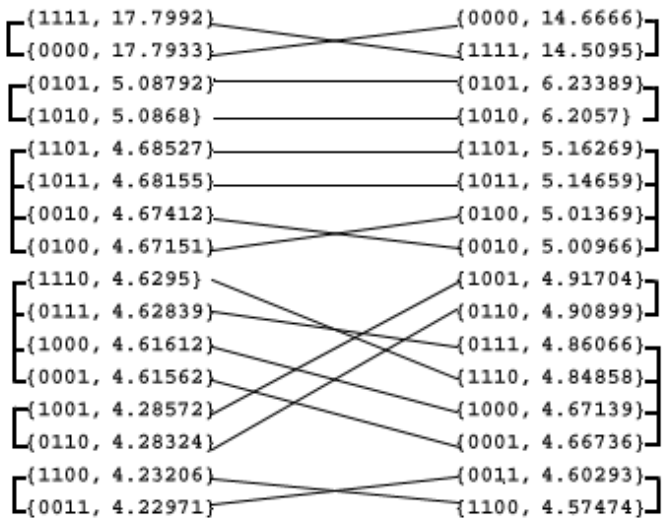

Figure 2.1: The above diagram shows the convergence of the frequency distributions of the outputs of TM and ECA for $k = 4$. Matching strings are linked by a line. As one can observe, in spite of certain crossings, TM and ECA are strongly correlated and both successfully group equivalent output strings. By taking the six groups–marked with brackets–the distribution frequencies only differ by one.

as two metrics for measuring the convergence of $TM(2,2)$ and $ECA(1,1)$ and the real-world data frequency distributions extracted from several sources [3]. A paper with mathematical formulations and further precise conjectures is currently in preparation.

Bibliography

[1] G.J. Chaitin, *Algorithmic Information Theory*, Cambridge University Press, 1987.

[2] G.J. Chaitin, *Information, Randomness and Incompleteness*, World Scientific, 1987.

[3] G.J. Chaitin, *Meta-Math! The Quest for Omega*, Pantheon Books NY, 2005.

[4] C.S. Calude, *Information and Randomness: An Algorithmic Perspective (Texts in Theoretical Computer Science. An EATCS Series)*, Springer; 2nd. edition, 2002.

[5] Kirchherr, W., M. Li, and P. Vitányi. The miraculous universal distribution. *Math. Intelligencer* 19(4), 7-15, 1997.

[6] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, Springer, 1997.

[7] A.K.Zvonkin, L. A. Levin. "The Complexity of finite objects and the Algorithmic Concepts of Information and Randomness.", *UMN = Russian Math. Surveys*, 25(6):83-124, 1970.

[8] S. Lloyd, *Programming the Universe*, Knopf, 2006.

[9] R. Solomonoff, The Discovery of Algorithmic Probability, *Journal of Computer and System Sciences*, Vol. 55, No. 1, pp. 73-88, August 1997.

[10] R. Solomonoff, *A Preliminary Report on a General Theory of Inductive Inference*, (Revision of Report V-131), Contract AF 49(639)-376, Report ZTB-138, Zator Co., Cambridge, Mass., Nov, 1960

[11] S. Wolfram, *A New Kind of Science*, Wolfram Media, 2002.

3. A website with the complete results of the whole experiment is available at http://www.mathrix.org/experimentalAIT

Chapitre 3

A Glance into the Structure of Algorithmic Randomness

From J.-P. Delahaye and H. Zenil, *Numerical Evaluation of Algorithmic Complexity for Short Strings: A Glance Into the Innermost Structure of Randomness*, 2011.

3.1 Introduction

We describe a method that combines several theoretical and experimental results to numerically approximate the algorithmic (Kolmogorov-Chaitin) complexity of all $\sum_{n=1}^{8} 2^n$ bit strings up to 8 bits long, and for some between 9 and 16 bits long. This is done by an exhaustive execution of all $(4n + 2)^{2n}$ deterministic 2-symbol Turing machines with up to $n = 4$ states for which the halting times are known thanks to the Busy Beaver problem. An output frequency distribution is then computed, from which the algorithmic probability is calculated and the algorithmic complexity evaluated by way of the (Levin-Chaitin) coding theorem.

The algorithmic complexity of a bit string is defined as the length of the shortest binary computer program that prints out the string (hence also called *program-size* complexity). However, no general, finite and deterministic procedure exists to calculate algorithmic complexity. For a given string

there are infinite many programs producing it. The most common approach to calculate the algorithmic complexity of a string is the use of compression algorithms exploiting the regularities of the string and producing shorter compressed versions. Compression algorithms have been used over the years, with the Lempel-Ziv algorithm[12] being one of the most prominent examples. The result of a compressed version of a string is an upper bound of the algorithmic complexity (denoted by $C(s)$) of the string s.

In practice, it is a known problem that one cannot compress short strings, shorter, for example, than the length in bits of the compression program which is added to the compressed version of s, making the result (the program producing s) sensitive to the compressor choice and the parameters involved. However, short strings are quite often the kind of data encountered in many practical settings. While compressors' asymptotic behavior guarantees the eventual convergence to $C(s)$ thanks to the invariance theorem (to be enunciated later), measurements differ considerably in the domain of short strings. A few attempts to deal with this problem have been reported before[22]. The conclusion is that estimators are always challenged by short strings.

Attempts to compute the uncomputable are always challenging, see for example [20, 1, 19] and more recently [7] and [8]. This often requires combining theoretical and experimental results. In this paper we describe a method to compute the algorithmic complexity (hereafter denoted by $C(s)$) of (short) bit strings by running a set of (relatively) large number of Turing machines for which the halting runtimes are known thanks to the Busy Beaver problem[20]. The method describes a way to find the shortest program given a standard formalism of Turing machines, executing all machines from the shortest (in number of states) to a certain (small) size one by one recording how many of them produce a string and then using a theoretical result linking this string frequency with the algorithmic complexity of a string that we use to approximate $C(s)$.

The approach we adopt here is different and independent of the machine size (small machines are used only because they allow us to calculate all of them up to a small size) and relies only on the concept of algorithmic probability. The exhaustive approach is in the lines of Wolfram's paradigm[24] consisting of exploring the space of simple programs by way of computer experiments. The result is a novel approach that we put forward for numerically calculate the complexity of short strings as an alternative to the indirect method using compression algorithms. The procedure makes use of a combination of results from related areas of computation, such as the concept of halting probability[3], the Busy Beaver problem[20], algorithmic probability[21],

Levin's semi-measure and Levin-Chaitin's coding theorem[13].

This approach, never attempted before to the authors' knowledge, consists in the thorough execution of all 4-state, 2-symbol Turing machines (the exact model is described in 3.3) which, upon halting, generate a set of output strings from which a frequency distribution is calculated to obtain the algorithmic probability of a string to be the result of a halting machine. The algorithmic complexity of a string is then evaluated from the algorithmic probability using Levin-Chaitin's coding theorem.

The paper is structured as follows. In section 3.2 it is introduced the various theoretical concepts and experimental results utilized in the experiment, providing essential definitions and referring the reader to the relevant papers and textbooks. Section 3.3 introduces the definition of our empirical probability distribution D. In 5.3 we present the methodology for calculating D. In 5.4 we calculate D and provide numerical values of the algorithmic complexity for short strings (the probability distribution produced by all 4-state 2-symbol Turing machines) by way of the theory presented in 3.2, particularly the Levin-Chaitin coding theorem. Finally, in 8.9 we summarize, discuss possible applications, and suggest potential directions for further research.

3.2 Preliminaries

3.2.1 The Halting problem and Chaitin's Ω

The Halting problem for Turing machines is the problem of deciding whether an arbitrary Turing machine T eventually halts on an arbitrary input s. One can ask whether there is a Turing machine U which, given $code(T)$ and the input s, eventually stops and produces 1 if $T(s)$ halts, and 0 if $T(s)$ does not halt. Turing's result[23] proves that there is no such U. A Cantor diagonalization argument[23] shows that the set of all such functions is not enumerable, whereas the set of all Turing machines is enumerable. Therefore, there are functions that are noncomputable.

Halting computations can be recognized by simply running them for the time they take to halt. The problem is to detect non-halting programs, about which one cannot know if the program will run forever or will eventually halt.

An elegant and concise representation of the halting problem is Chaitin's irrational number Ω[3], defined as the halting probability of a universal computer programmed by coin tossing. Formally,

DEFINITION 1. $0 < \Omega = \sum_{p\ halts} 2^{-|p|} < 1$ with $|p|$ the size of p in bits.

Ω is the halting probability of a universal (prefix-free[1]) Turing machine running a random program[2]. From now on only plain complexity will be used in this paper, unless explicitly stated otherwise.

In contrast with the mathematical constant π, for which, given enough time, one can in principle compute any number of digits of its binary expansion, for an Ω number one cannot compute more than a finite number of digits. The numerical value of $\Omega = \Omega_U$ depends on the choice of universal Turing machine U. There are, for example, Ω numbers for which no digit can be computed[3]. For a Ω number it matters only whether a program halts or not; the time at which a halting program stops and the output upon halting is irrelevant.

A program that halts will eventually halt and its contribution to Ω can be recorded. Knowing the first n bits of Ω allows to determine whether a program of length $\leq n$ bits halts by simply running all programs in parallel until the sum exceeds Ω_n. All programs with length $\leq n$ not halting yet will not halt. Using these results, Calude and Stay[6] have shown that most programs either stop "quickly" or never halt because the halting runtime (and therefore the length of the output upon halting) is ultimately bounded by its program-size complexity. The results herein connect theory with experiments by providing empirical values of halting times and string length frequencies.

3.2.2 Algorithmic (program-size) complexity

The algorithmic complexity $C_U(s)$ of a string s with respect to a universal Turing machine U, measured in bits, is defined as the length in bits of the shortest Turing machine U that produces the string s and halts[21, 11, 13, 3]. Formally,

DEFINITION 2. $C_U(s) = \min\{|p|, U(p) = s\}$ where $|p|$ is the length of p measured in bits.

1. A set of programs A is prefix-free if there are no two programs p_1 and p_2 such that p_2 is a proper extension of p_1. Kraft's inequality[?] guarantees that for any prefix-free set A, $\sum_{x \in A} 2^{-|x|} \leq 1$.

2. The differences and compatibilities between plain and prefix-free complexity are studied in detail in basic texts such as [?, 16].

This complexity measure clearly seems to depend on U, and one may ask whether there exists a Turing machine which yields different values of $C(s)$. The answer is that there is no such Turing machine which can be used to decide whether a short description of a string is the shortest (for formal proofs see [**?**, 16]).

The ability of universal machines to efficiently simulate each other implies a corresponding degree of robustness. The invariance theorem[21] states that if $C_U(s)$ and $C_{U'}(s)$ are the shortest programs generating s using the universal Turing machines U and U' respectively, their difference will be bounded by an additive constant independent of s. Formally:

THEOREM (INVARIANCE[21]) 1. $|C_U(s) - C_{U'}(s)| \leq c_{U,U'}$

It is easy to see that the underlying concept is that, since both U and U' are universal Turing machines, one can always write a general translator (a compiler) between U and U' such that one can run either Turing machine and get one or another complexity value, simply adding the constant length of the translator to the result.

Algorithmic complexity formalizes the concept of simplicity versus complexity. It opposes what is simple to what is complex or random. Chaitin proves, for example, that his Ω numbers are algorithmically random. Unlike a simple string, a random one is hard to describe and only long programs, of about the original string length, can produce it.

A major drawback of C as a function taking s to the length of the shortest program producing s, is its non-computability proven by reduction to the halting problem. In other words, there is no program which takes a string s as input and produces the integer $C(s)$ as output.

3.2.3 Algorithmic probability

Deeply connected to Chaitin's halting probability Ω, is Solomonoff's concept of algorithmic probability, independently proposed and further formalized by Levin's[13] semi-measure herein denoted by $m(s)$.

Unlike Chaitin's Ω, it is not only whether a program halts or not that matters for the concept of algorithmic probability; the output and halting time of a halting Turing machine are also relevant in this case.

Levin's semi-measure $m(s)$ is the probability of producing a string s with a random program p when running on a universal prefix-free Turing

machine U. Formally,

DEFINITION 3. $m(s) = \sum_{p:U(p)=s} 2^{-|p|}$

In other words, $m(s)$ is the sum over all the programs for which the (prefix-free) universal Turing machine U outputs the string s and halts, when provided with a sequence of fair coin flip inputs as a program p. Levins probability measure induces a distribution over programs producing s, assigning to the shortest program the highest probability and smaller probabilities to longer programs. $m(s)$ indicates that a string s is likely to be produced by the shortest program producing s.

m is related to algorithmic complexity in that $m(s)$ is at least the maximum term in the summation of programs given that the shortest program has the greater weight in the addition. So by calculating $m(s)$, one can obtain $C(s)$.

THEOREM (CODING THEOREM[5]) 2. $-\log_2 m(s) = C(s) + O(1)$

The coding theorem states that the shortest description of a given binary string is the negative logarithm of its algorithmic probability up to an additive constant.

Nevertheless, $m(s)$ as a function of s is, like $C(s)$ and Chaitin's Ω, non-computable due to the halting problem [3].

Levin's semi-measure$m(s)$ formalizes the concept of Occams razor, that is, the principle that favors selecting the hypothesis (the generating program) that makes the fewest assumptions (the shortest) when the set of competing hypotheses (all programs producing the same output) are equal in all other respects.

3.2.4 The Busy Beaver problem

NOTATION: We denote by (n,2) the class (or space) of all n-state 2-symbol Turing machines (with the halting state not included among the n states).

3. An important property of m as semi-measure is that it dominates any other effective semi-measure μ because there is a constant c_μ such that, for all s $m(s) \geq c_\mu \mu(s)$. For this reason $m(s)$ is often called a *universal distribution*[10].

Definition 4.[20] If σ_T is the number of 1s on the tape of a Turing machine T upon halting, then: $\sum(n) = \max\{\sigma_T : T \in (n,2)\ T(n)\ halts\}$.

Definition 5.[20] If t_T is the number of steps that a machine T takes upon halting, then $S(n) = \max\{t_T : T \in (n,2)\ T(n)\ halts\}$.

$\sum(n)$ and $S(n)$ as defined in 4 and 5 are noncomputable by reduction to the halting problem. Yet values are known for (n,2) with $n \leq 4$. The solution for (n,2) with $n < 3$ is trivial, the process leading to the solution in (3,2) is discussed by Lin and Rado[17], and the process leading to the solution in (4,2) is discussed in [1].

A program showing the evolution of all known Busy Beaver machines developed by one of this paper's authors is available online[27]. The formalism followed in this paper is exactly the same as the one originally described and followed for the Busy Beaver problem as introduced by Rado[20] .

Solving the halting problem for small machines

The halting problem and the halting probability problem are closely related to the Busy Beaver problem in that a solution to any one of them would yield a solution to each of the others.

It is easy to see that $\sum(1) = 1$ and $\sum(2) = 4$. Lin and Rado[17] proved $\sum(3) = 6$ and [1] $\sum(4) = 13$. The exact known values for S are $S(1) = 1$, $S(2) = 6$, $S(3) = 21$, $S(4) = 107$. These Busy Beaver values are for 2-symbol Turing machines.

3.3 The empirical distribution D

Definition 6. Consider a Turing machine with the binary alphabet $\Sigma = \{0, 1\}$ and n states $\{1, 2, \ldots n\}$ and an additional Halt state denoted by 0 (just as defined in Rado's original Busy Beaver paper[20]).

The machine runs on a 2-way unbounded tape. At each step:

1. the machine's current "state" (instruction); and
2. the tape symbol the machine's head is scanning

define each of the following:

1. a unique symbol to write (the machine can overwrite a 1 on a 0, a 0 on a 1, a 1 on a 1, and a 0 on a 0);
2. a direction to move in: -1 (left), 1 (right) or 0 (none, when halting); and
3. a state to transition into (may be the same as the one it was in).

The machine halts if and when it reaches the special halt state 0.

Since the domain of the program has size $2n$ and the target space has size $4n + 2$, we can easily count the number of Turing machines. There are $(4n + 2)^{2n}$ Turing machines with n states and 2 symbols according to the formalism described above.

No transition starting from the halting state exists, and the blank symbol is one of the 2 symbols (0 or 1) in the first run, while the other is used in the second run (in order to avoid any asymmetries due to the choice of a single blank symbol). In other words, we run each machine twice, one with 0 as the blank symbol (the symbol with which the tape starts out and is filled with), and an additional run with 1 as the blank symbol [4].

Definition 7. $D(n)$ = the function that assigns to every finite bit string s the quotient (number of times that a machine (n,2) produces s) / (number of machines in (n,2)).

The output string is taken from the number of contiguous cells on the tape the head of the halting n-state machine has gone through. A machine produces a string upon halting.

Examples of $D(n)$ for $n = 1, n = 2$:

$$D(1) = 0 \to 0.5; 1 \to 0.5$$
$$D(2) = 0 \to 0.328; 1 \to 0.328; 00 \to .0834\ldots$$

$D(n)$ is the probability distribution of the strings produced by all 2-symbol halting Turing machines with n states. Tables 1, 2 and 3 in 5.4 show

4. Due to the symmetry of the computation, there is no real need to run each machine twice; one can *complete* the string frequencies assuming that each string produced its reversed and complemented version with the same frequency, and then group and divide by symmetric groups. A more detailed explanation of how this is done is in [2].

the full results for $D(1)$, $D(2)$ and $D(3)$, and Table 4 the top ranking of $D(4)$.

THEOREM 3. $D(n)$ is not computable.

PROOF (BY CONTRADICTION TO REDUCTION TO THE BUSY BEAVER PROBLEM) Suppose that $n \to D(n)$ is computable. Let n be a given integer. Let $D(n)$ be computed. By definition $D(n)$ is the function that assigns to every finite bit string s the quotient (number of times that a machine (n,2) produces s upon halting) / (number of machines in (n,2)). To calculate D we use the values of the Busy Beaver function $S(n)$. Turing machines that run more than $S(n)$ steps will therefore never halt. Computing $D(n)$ for any n would therefore mean that the Busy Beaver functions $S(n)$ and $\Sigma(n)$ can be known for any n providing a method to calculate Rado's Busy Beaver functions. Which is a contradiction with the non-computability of Rado's Busy Beaver functions.

Exact values can be, however, calculated for small Turing machines up to $n = 4$ states thanks to the Busy Beaver values of $S(n)$ for $n < 5$. For example, for $n = 4$, $S(4) = 107$, so 107 is the maximum runtime we have to run each machine in (4,2) in order to get the all the outputs.

For each Busy Beaver candidate with $n > 4$ states, a sample of Turing machines running up to the candidate $S(n)$ is possible. As for Rado's Busy Beaver functions $\sum(n)$ and $S(n)$, D is also approachable *from above*. For larger n sampling methods asymptotically converging to $D(n)$ can be used to approximate $D(n)$. In 5.4 we provide exact values of $D(n)$ for $n < 5$ for which the Busy Beaver functions are known.

Another common property between $D(n)$ and Rado's Busy Beaver functions is that $D(4)$ is well-defined in the sense that the calculation of the digits of $D(n)$ are fully determined as is in the decimal expansion of the mathematical constant π, but the calculation of $D(n)$ rapidly becomes impractical to determine for even a slightly larger number of states.

3.4 Methodology

The approach for evaluating the complexity $C(s)$ of a string s presented herein is limited by (1) the halting problem and (2) computing time constraints. Restriction (1) was overcome because the values of the Busy Beaver problem provided the halting times for all (4,2) Turing machines that started

with a blank tape. Restriction (2) represented a challenge in terms of computing time and programming skills. It is also the same restriction that has kept others from attempting to solve the Busy Beaver problem for a greater number of states. Constraint (2) and the knowledge of the values of the Busy Beaver function permitted us to systematically study machines up to 4 states. We were able to compute up to about 1.3775×10^9 machines per day or 15 943 per second, taking us about 9 days[5] to run all (4,2) Turing machines each up to the number of steps bounded by the Busy Beaver values.

Our quest is similar in several respects to the Busy Beaver problem or the calculation of the digits of Chaitin's Ω number. The main underlying difficulty in analyzing thoroughly a given class of machines is their uncomputability. Just as it is done for solving small values of the Busy Beaver problem, we rely on the experimental approach to analyze and describe a computable fraction of the uncomputable. A similar quest for the calculation of the digits of a Chaitin's Ω number was undertaken by Calude et al.[7], but unlike Chaitin's Ω, the calculation of $D(n)$ does not depend on the enumeration of Turing machines. In $D(n)$, however, one obtains different probabilities for the same string for each n, but the relative order seems to be preserved. In fact, every $(2, n)$ machine contributing to $D(n)$ is included in $D(n+1)$ simply because every machine rule in $(2, n)$ is in $(2, n+1)$.

3.4.1 Numerical calculation of D

We consider the space (n,2) of Turing machines with $0 < n < 5$. The halting "history" and output probability followed by their respective runtimes, presented in Tables 1, 2 and 3, show the times at which the programs in the domain of M halt, the frequency of the strings produced, and the time at which they halted after writing down the output string on their tape.

We provide exact values for $n = \{2, 3, 4\}$ in the Results 5.4. We derive $D(n)$ for $n < 5$ from counting the number of n-strings produced by all (n,2) Turing machines upon halting. We define D to be an *empirical universal distribution* in Levin's sense, and calculate the algorithmic complexity C of a string s in terms of D using the coding theorem, from which we won't

5. Running on a single computer on a MacBook Intel Core Duo at 1.83Ghz, 2Gb. of memory and a solid state hard drive, using the TuringMachine[] function available in *Mathematica* 8 for $n < 4$ and a C++ program for $n = 4$. Since for $n = 4$ there were 2.56×10^8 machines involved, running on both 0 and 1 as blank, further optimizations were required. The use of a bignum library and an actual enumeration of the machines rather than producing the rules beforehand (which would have meant overloading the memory even before the actual calculation) was necessary.

escape to an additive constant introduced by the application of the coding theorem, but the additive constant is common to all values and therefore should not impact the relative order. One has to bear in mind, however, that the tables in section 5.4 should be read as dependent of this last-step additive constant because using the coding theorem as an approximation method fixes a prefix-free UTM via that constant, but according to the choices we make this seems to be the most natural way to do so as an alternative to other indirect numerical methods.

We calculated the 72, 20 000, 15 059 072 and 22 039 921 152 two-way tape Turing machines started with a tape filled with 0s and 1s for $D(2)$, $D(3)$ and $D(4)$ [6]. The number of Turing machines to calculate grows exponentially with the number of states. For $D(5)$ there are 53 119 845 582 848 machines to calculate, which makes the task as difficult as finding the Busy Beaver values for $\sum(5)$ and $S(5)$, Busy Beaver values which are currently unknown but for which the best candidate may be $S(5) = 47\,176\,870$ which makes the exploration of (5,2) quite a challenge.

Although several ideas exploiting symmetries to reduce the total number of Turing machines have been proposed and used for finding Busy Beaver candidates[1, 18, 9] in large spaces such as $n \geq 5$, to preserve the structure of the data we couldn't apply all of them. This is because, unlike the Busy Beaver challenge, in which only the maximum values are important, the construction of a probability distribution requires every output to be equally considered. Some reduction techniques were, however, utilized, such as running only one-direction rules with a tape only filled with 0s and then completing the strings by reversion and complementation to avoid running every machine a second time with a tape filled with 1s. For an explanation of how we counted the number of symmetries to recuperate the outputs of the machines that were skipped see [2].

6. We ran the experiment on several computers and cores in parallel, which allowed us to shorten the time by about a fourth of that calculated in a single processor. The space occupied by the machine outputs was 77.06 GB (of which only 38.53 GB was actually necessary by taking advantage of machine rule symmetries that could be later compensated without having to run them).

3.5 Results

3.5.1 Algorithmic probability tables

$D(1)$ is trivial. (1,2) Turing machines produce only two strings, with the same number of machines producing each. The Busy Beaver values for $n = 1$ are $\sum(1) = 1$ and $S(1) = 1$. That is, all machines that halt do so after 1 step, and print at most one symbol.

Table 3.1: Complete $D(1)$ from 24 (1,2)-Turing machines that halt out of a total of 64.

$0 \to 0.5$
$1 \to 0.5$

The Busy Beaver values for $n = 2$ are $\sum(1) = 4$ and $S(1) = 6$. $D(2)$ is quite simple but starts to display some basic structure, such as a clear correlation between string length and occurrence, following what may be an exponential decrease:

$$P(|s| = 1) = 0.657$$
$$P(|s| = 2) = 0.333$$
$$P(|s| = 3) = 0.0065$$
$$P(|s| = 4) = 0.0026$$

Among the various facts one can draw from $D(2)$, there are:

- The relative string order in $D(1)$ is preserved in $D(2)$.
- A fraction of 1/3 of the total machines halt while the remaining 2/3 do not. That is, 24 among 72 (running each machine twice with tape filled with 1 and 0 as explained before).
- The longest string produced by $D(2)$ is of length 4.
- $D(2)$ does not produce all $\sum_1^4 2^n = 30$ strings shorter than 5, only 22. The missing strings are 0001, 0101 and 0011 never produced, hence neither were their complements and reversions: 0111, 1000, 1110, 1010 and 1100.

Given the number of machines to run, $D(3)$ constitutes the first non trivial probability distribution to calculate. The Busy Beaver values for $n = 3$ are $\sum(3) = 6$ and $S(3) = 14$.

Table 3.2: Complete $D(2)$ (22 bit-strings) from 6 088 (2,2)-Turing machines that halt out of 20 000. Each string is followed by its probability (from the number of times produced), sorted from highest to lowest.

$0 \to .328$	$010 \to .00065$
$1 \to .328$	$101 \to .00065$
$00 \to .0834$	$111 \to .00065$
$01 \to .0834$	$0000 \to .00032$
$10 \to .0834$	$0010 \to .00032$
$11 \to .0834$	$0100 \to .00032$
$001 \to .00098$	$0110 \to .00032$
$011 \to .00098$	$1001 \to .00032$
$100 \to .00098$	$1011 \to .00032$
$110 \to .00098$	$1101 \to .00032$
$000 \to .00065$	$1111 \to .00032$

Among the various facts for $D(3)$:

- There are 4 294 368 machines that halt among the 15 059 072 in (3,2). That is a fraction of 0.2851.
- The longest string produced in (3,2) is of length 7.
- $D(3)$ has not all $\sum_1^7 2^n = 254$ strings shorter than 7 but 128 only, half of all the possible strings up to that length.
- $D(3)$ preserves the string order of $D(2)$.

$D(3)$ ratifies the tendency of classifying strings by length with exponentially decreasing values. The distribution comes sorted by length blocks from which one cannot easily say whether those at the bottom are more random-looking than those in the middle, but one can definitely say that the ones at the top, both for the entire distribution and by length block, are intuitively the simplest. Both 0^k and its reversed 1^k for $n \leq 8$ are always at the top of each block, with 0 and 1 at the top of them all. There is a single exception in which strings were not sorted by length, this is the string group 0101010 and 1010101 that are found four places further away from their length block, which we take as a second indication of a complexity classification becoming more visible since these 2 strings correspond to what one would intuitively consider less random-looking because they are easily described as the repetition of two bits.

$D(4)$ with 22 039 921 152 machines to run was the first true challenge, both in terms of programming specification and computational resources.

Table 3.3: Complete $D(3)$ (128 bit-strings) produced by all the 15 059 072 (3,2)-halting Turing machines.

$0 \to 0.250$	$11110 \to 0.0000470$	$100101 \to 1.43\times10^{-6}$
$1 \to 0.250$	$00100 \to 0.0000456$	$101001 \to 1.43\times10^{-6}$
$00 \to 0.101$	$11011 \to 0.0000456$	$000011 \to 9.313\times10^{-7}$
$01 \to 0.101$	$01010 \to 0.0000419$	$000110 \to 9.313\times10^{-7}$
$10 \to 0.101$	$10101 \to 0.0000419$	$001100 \to 9.313\times10^{-7}$
$11 \to 0.101$	$01001 \to 0.0000391$	$001101 \to 9.313\times10^{-7}$
$000 \to 0.0112$	$01101 \to 0.0000391$	$001111 \to 9.313\times10^{-7}$
$111 \to 0.0112$	$10010 \to 0.0000391$	$010001 \to 9.313\times10^{-7}$
$001 \to 0.0108$	$10110 \to 0.0000391$	$010010 \to 9.313\times10^{-7}$
$011 \to 0.0108$	$01110 \to 0.0000289$	$010011 \to 9.313\times10^{-7}$
$100 \to 0.0108$	$10001 \to 0.0000289$	$011000 \to 9.313\times10^{-7}$
$110 \to 0.0108$	$00101 \to 0.0000233$	$011101 \to 9.313\times10^{-7}$
$010 \to 0.00997$	$01011 \to 0.0000233$	$011110 \to 9.313\times10^{-7}$
$101 \to 0.00997$	$10100 \to 0.0000233$	$100001 \to 9.313\times10^{-7}$
$0000 \to 0.000968$	$11010 \to 0.0000233$	$100010 \to 9.313\times10^{-7}$
$1111 \to 0.000968$	$00011 \to 0.0000219$	$100111 \to 9.313\times10^{-7}$
$0010 \to 0.000699$	$00111 \to 0.0000219$	$101100 \to 9.313\times10^{-7}$
$0100 \to 0.000699$	$11000 \to 0.0000219$	$101101 \to 9.313\times10^{-7}$
$1011 \to 0.000699$	$11100 \to 0.0000219$	$101110 \to 9.313\times10^{-7}$
$1101 \to 0.000699$	$000000 \to 3.733\times10^{-6}$	$110000 \to 9.313\times10^{-7}$
$0101 \to 0.000651$	$111111 \to 3.733\times10^{-6}$	$110010 \to 9.313\times10^{-7}$
$1010 \to 0.000651$	$000001 \to 2.793\times10^{-6}$	$110011 \to 9.313\times10^{-7}$
$0001 \to 0.000527$	$011111 \to 2.793\times10^{-6}$	$111001 \to 9.313\times10^{-7}$
$0111 \to 0.000527$	$100000 \to 2.793\times10^{-6}$	$111100 \to 9.313\times10^{-7}$
$1000 \to 0.000527$	$111110 \to 2.793\times10^{-6}$	$0101010 \to 9.313\times10^{-7}$
$1110 \to 0.000527$	$000100 \to 2.333\times10^{-6}$	$1010101 \to 9.313\times10^{-7}$
$0110 \to 0.000510$	$001000 \to 2.333\times10^{-6}$	$001110 \to 4.663\times10^{-7}$
$1001 \to 0.000510$	$110111 \to 2.333\times10^{-6}$	$011100 \to 4.663\times10^{-7}$
$0011 \to 0.000321$	$111011 \to 2.333\times10^{-6}$	$100011 \to 4.663\times10^{-7}$
$1100 \to 0.000321$	$000010 \to 1.863\times10^{-6}$	$110001 \to 4.663\times10^{-7}$
$00000 \to 0.0000969$	$001001 \to 1.863\times10^{-6}$	$0000010 \to 4.663\times10^{-7}$
$11111 \to 0.0000969$	$001010 \to 1.863\times10^{-6}$	$0000110 \to 4.663\times10^{-7}$
$00110 \to 0.0000512$	$010000 \to 1.863\times10^{-6}$	$0100000 \to 4.663\times10^{-7}$
$01100 \to 0.0000512$	$010100 \to 1.863\times10^{-6}$	$0101110 \to 4.663\times10^{-7}$
$10011 \to 0.0000512$	$011011 \to 1.863\times10^{-6}$	$0110000 \to 4.663\times10^{-7}$
$11001 \to 0.0000512$	$100100 \to 1.863\times10^{-6}$	$0111010 \to 4.663\times10^{-7}$
$00010 \to 0.0000489$	$101011 \to 1.863\times10^{-6}$	$1000101 \to 4.663\times10^{-7}$
$01000 \to 0.0000489$	$101111 \to 1.863\times10^{-6}$	$1001111 \to 4.663\times10^{-7}$
$10111 \to 0.0000489$	$110101 \to 1.863\times10^{-6}$	$1010001 \to 4.663\times10^{-7}$
$11101 \to 0.0000489$	$110110 \to 1.863\times10^{-6}$	$1011111 \to 4.663\times10^{-7}$
$00001 \to 0.0000470$	$111101 \to 1.863\times10^{-6}$	$1111001 \to 4.663\times10^{-7}$
$01111 \to 0.0000470$	$010110 \to 1.43\times10^{-6}$	$1111101 \to 4.663\times10^{-7}$
$10000 \to 0.0000470$	$011010 \to 1.43\times10^{-6}$	

The Busy Beaver values for $n = 4$ are $\sum(3) = 13$ and $S(n) = 107$. Evidently every machine in (n,2) for $n \leq 4$ is in (4,2) because a rule in (n,2) with $n \leq 4$ is a rule in (4,2) in which a part of it is never used and halts, which is guaranteed to exist because the computation is exhaustive over (4,2).

Among the various facts from these results:

- There are 5 970 768 960 machines that halt among the 22 039 921 152 in (4,2). That is a fraction of 0.27.
- A total number of 1824 strings were produced in (4,2).
- The longest string produced is of length 16 (only 8 among all the 2^{16} possible were generated).
- The Busy Beaver machines (writing more 1s than any other and halting) found in (4,2) had very low probability among all the halting machines: $pr(11111111111101) = 2.01 \times 10^{-9}$. Because of the reverted string (10111111111111), the total probability of finding a Busy Beaver in (4,2) is therefore 4.02×10^{-9} only (or twice that number if the complemented string with the maximum number of 0s is taken).
- The longest strings in (4,2) were in the string group formed by: 1101010 101010101, 1101010100010101, 1010101010101011 and 1010100010101011, each with 5.4447×10^{-10} probability, i.e. an even smaller probability than for the Busy Beavers, and therefore the most random in the classification.
- (4,2) produces all strings up to length 8, then the number of strings larger than 8 decreases. The following are the number of strings by length $|\{s : |s| = l\}|$ generated and represented in $D(4)$ from a total of 1 824 different strings. From $i = 1, \ldots, 15$ the values l of $|\{s : |s| = n\}|$ are 2, 4, 8, 16, 32, 64, 128, 256, 486, 410, 252, 112, 46, 8, and 0, which indicated all 2^l strings where generated for $n \leq 8$.
- While the probability of producing a string with an odd number of 1s is the same than the probability of producing a string with an even number of 1s (and therefore the same for 0s), the probability of producing a string of odd length is .559 and .441 for even length.
- As in $D(3)$, where we report that one string group (0101010 and its reversion), in $D(4)$ 399 strings climbed to the top and were not sorted among their length groups.
- In $D(4)$ string length was no longer a determinant for string positions. For example, between positions 780 and 790, string lengths are: 11, 10, 10, 11, 9, 10, 9, 9, 9, 10 and 9 bits.
- $D(4)$ preserves the string order of $D(3)$ except in 17 places out of 128

Table 3.4: The top 129 strings from $D(4)$ with highest probability from a total of 1832 different produced strings.

$0 \to 0.205$	$01101 \to 0.000145$	$110111 \to 0.0000138$
$1 \to 0.205$	$10010 \to 0.000145$	$111011 \to 0.0000138$
$00 \to 0.102$	$10110 \to 0.000145$	$001001 \to 0.0000117$
$01 \to 0.102$	$01010 \to 0.000137$	$011011 \to 0.0000117$
$10 \to 0.102$	$10101 \to 0.000137$	$100100 \to 0.0000117$
$11 \to 0.102$	$00110 \to 0.000127$	$110110 \to 0.0000117$
$000 \to 0.0188$	$01100 \to 0.000127$	$010001 \to 0.0000109$
$111 \to 0.0188$	$10011 \to 0.000127$	$011101 \to 0.0000109$
$001 \to 0.0180$	$11001 \to 0.000127$	$100010 \to 0.0000109$
$011 \to 0.0180$	$00101 \to 0.000124$	$101110 \to 0.0000109$
$100 \to 0.0180$	$01011 \to 0.000124$	$000011 \to 0.0000108$
$110 \to 0.0180$	$10100 \to 0.000124$	$001111 \to 0.0000108$
$010 \to 0.0171$	$11010 \to 0.000124$	$110000 \to 0.0000108$
$101 \to 0.0171$	$00011 \to 0.000108$	$111100 \to 0.0000108$
$0000 \to 0.00250$	$00111 \to 0.000108$	$000110 \to 0.0000107$
$1111 \to 0.00250$	$11000 \to 0.000108$	$011000 \to 0.0000107$
$0001 \to 0.00193$	$11100 \to 0.000108$	$100111 \to 0.0000107$
$0111 \to 0.00193$	$01110 \to 0.0000928$	$111001 \to 0.0000107$
$1000 \to 0.00193$	$10001 \to 0.0000928$	$001101 \to 0.0000101$
$1110 \to 0.00193$	$000000 \to 0.0000351$	$010011 \to 0.0000101$
$0101 \to 0.00191$	$111111 \to 0.0000351$	$101100 \to 0.0000101$
$1010 \to 0.00191$	$000001 \to 0.0000195$	$110010 \to 0.0000101$
$0010 \to 0.00190$	$011111 \to 0.0000195$	$001100 \to 9.943\times10^{-6}$
$0100 \to 0.00190$	$100000 \to 0.0000195$	$110011 \to 9.943\times10^{-6}$
$1011 \to 0.00190$	$111110 \to 0.0000195$	$011110 \to 9.633\times10^{-6}$
$1101 \to 0.00190$	$000010 \to 0.0000184$	$100001 \to 9.633\times10^{-6}$
$0110 \to 0.00163$	$010000 \to 0.0000184$	$011001 \to 9.3\times10^{-6}$
$1001 \to 0.00163$	$101111 \to 0.0000184$	$100110 \to 9.3\times10^{-6}$
$0011 \to 0.00161$	$111101 \to 0.0000184$	$000101 \to 8.753\times10^{-6}$
$1100 \to 0.00161$	$010010 \to 0.0000160$	$010111 \to 8.753\times10^{-6}$
$00000 \to 0.000282$	$101101 \to 0.0000160$	$101000 \to 8.753\times10^{-6}$
$11111 \to 0.000282$	$010101 \to 0.0000150$	$111010 \to 8.753\times10^{-6}$
$00001 \to 0.000171$	$101010 \to 0.0000150$	$001110 \to 7.863\times10^{-6}$
$01111 \to 0.000171$	$010110 \to 0.0000142$	$011100 \to 7.863\times10^{-6}$
$10000 \to 0.000171$	$011010 \to 0.0000142$	$100011 \to 7.863\times10^{-6}$
$11110 \to 0.000171$	$100101 \to 0.0000142$	$110001 \to 7.863\times10^{-6}$
$00010 \to 0.000166$	$101001 \to 0.0000142$	$001011 \to 6.523\times10^{-6}$
$01000 \to 0.000166$	$001010 \to 0.0000141$	$110100 \to 6.523\times10^{-6}$
$10111 \to 0.000166$	$010100 \to 0.0000141$	$000111 \to 6.243\times10^{-6}$
$11101 \to 0.000166$	$101011 \to 0.0000141$	$111000 \to 6.243\times10^{-6}$
$00100 \to 0.000151$	$110101 \to 0.0000141$	$0000000 \to 3.723\times10^{-6}$
$11011 \to 0.000151$	$000100 \to 0.0000138$	$1111111 \to 3.723\times10^{-6}$
$01001 \to 0.000145$	$001000 \to 0.0000138$	$0101010 \to 2.393\times10^{-6}$

strings in $D(3)$ ordered from highest to lowest string frequency. The maximum rank distance among the farthest two differing elements in $D(3)$ and $D(4)$ was 20, with an average of 11.23 among the 17 misplaced cases and a standard deviation of about 5 places. The Spearman's rank correlation coefficient between the two rankings had a critical value of 0.98, meaning that the order of the 128 elements in $D(3)$ compared to their order in $D(4)$ were in an interval confidence of high significance with almost null probability to have produced by chance.

Table 3.5: Probabilities of finding n 1s (or 0s) in $(4, 2)$.

number n of 1s	pr(n)
1	0.472
2	0.167
3	0.0279
4	0.00352
5	0.000407
6	0.0000508
7	6.5×10^{-6}
8	1.31×10^{-6}
9	2.25×10^{-7}
10	3.62×10^{-8}
11	1.61×10^{-8}
12	1.00×10^{-8}
13	4.02×10^{-9}

Same length string distribution

The following are the top 10 string groups (i.e. with their reverted and complemented counterparts) in $D(4)$ appearing sooner than expected and getting away from their length blocks. That is, their lengths were greater than the next string in the classification order): 11111111, 11110111, 000000000, 111111111, 000010000, 111101111, 111111110, 010101010, 101010101, 000101010. This means these string groups had greater algorithmic probability and therefore less algorithmic complexity than shorter strings.

Table 8 displays some statistical information of the distribution. The

Table 3.6: String groups formed by reversion and complementation followed by the total machines producing them.

string group	occurrences
0, 1	1224440064
01, 10	611436144
00, 11	611436144
001, 011, 100, 110	215534184
000, 111	112069020
010, 101	102247932
0001, 0111, 1000, 1110	23008080
0010, 0100, 1011, 1101	22675896
0000, 1111	14917104
0101, 1010	11425392
0110, 1001	9712752
0011, 1100	9628728
00001, 01111, 10000, 11110	2042268
00010, 01000, 10111, 11101	1984536
01001, 01101, 10010, 10110	1726704
00000, 11111	1683888
00110, 01100, 10011, 11001	1512888
00101, 01011, 10100, 11010	1478244
00011, 00111, 11000, 11100	1288908
00100, 11011	900768
01010, 10101	819924
01110, 10001	554304
000001, 011111, 100000, 111110	233064
000010, 010000, 101111, 111101	219552
000000, 111111	209436
010110, 011010, 100101, 101001	169896
001010, 010100, 101011, 110101	167964
000100, 001000, 110111, 111011	164520
001001, 011011, 100100, 110110	140280
010001, 011101, 100010, 101110	129972

Table 3.7: The probability of producing a string of length l exponentially decreases as l linearly increases. The slowdown in the rate of decrease for string length $l > 8$ is due to the few longer strings produced in (4,2).

length n	pr(n)
1	0.410
2	0.410
3	0.144
4	0.0306
5	0.00469
6	0.000818
7	0.000110
8	0.0000226
9	4.69×10^{-6}
10	1.42×10^{-6}
11	4.9×10^{-7}
12	1.69×10^{-7}

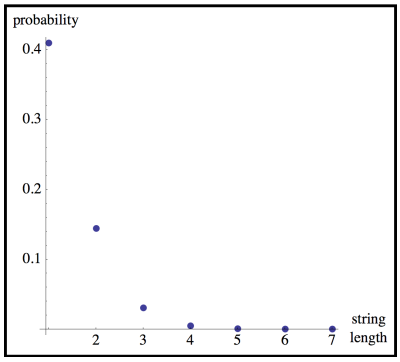

Figure 3.1: (4,2) frequency distribution by string length.

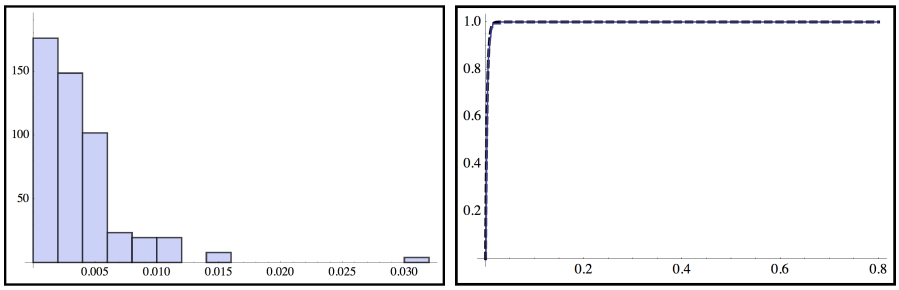

Figure 3.2: Probability density function of bit strings of length $l = 8$ from $(4, 2)$. The histogram (left) shows the probabilities to fall within a particular region. The cumulative version (right) shows how well the distribution fits a Pareto distribution (dashed) with location parameter $k = 10$. The reader may see but a single curve, that is because the lines overlap. $D(4)$ (and the sub-distributions it contains) is therefore log-normal.

distribution is skewed to the right, the mass of the distribution is therefore concentrated on the left with a long right tail, as shown in Figure 2.

Table 3.8: Statistical values of the empirical distribution function $D(4)$ for strings of length $l = 8$.

	value
mean	0.00391
median	0.00280
variance	0.0000136
kurtosis	23
skewness	3.6

3.5.2 Derivation and calculation of algorithmic complexity

Algorithmic complexity values are calculated from the output probability distribution $D(4)$ through the application of the coding theorem.

The largest complexity value $\max\{C(s) : s \in D(4)\} = 29$ bits. The output strings (11111111111101 and 10111111111111) produced by the Busy Beavers in (4,2) were close to the maximal complexity in this space for the

Table 3.9: Top 180 strings from $D(4)$ sorted from lowest to highest algorithmic complexity.

0→2.29	10110→12.76	100100→16.38	0100000→19.10
1→2.29	01010→12.83	110110→16.38	1011111→19.10
00→3.29	10101→12.83	010001→16.49	1111101→19.10
01→3.29	00110→12.95	011101→16.49	0000100→19.38
10→3.29	01100→12.95	100010→16.49	0010000→19.38
11→3.29	10011→12.95	101110→16.49	1101111→19.38
000→5.74	11001→12.95	000011→16.49	1111011→19.38
111→5.74	00101→12.98	001111→16.49	0001000→19.45
001→5.79	01011→12.98	110000→16.49	1110111→19.45
011→5.79	10100→12.98	111100→16.49	0000110→19.64
100→5.79	11010→12.98	000110→16.52	0110000→19.64
110→5.79	00011→13.18	011000→16.52	1001111→19.64
010→5.87	00111→13.18	100111→16.52	1111001→19.64
101→5.87	11000→13.18	111001→16.52	0101110→19.68
0000→8.64	11100→13.18	001101→16.59	0111010→19.68
1111→8.64	01110→13.39	010011→16.59	1000101→19.68
0001→9.02	10001→13.39	101100→16.59	1010001→19.68
0111→9.02	000000→14.80	110010→16.59	0010001→20.04
1000→9.02	111111→14.80	001100→16.62	0111011→20.04
1110→9.02	000001→15.64	110011→16.62	1000100→20.04
0101→9.03	011111→15.64	011110→16.66	1101110→20.04
1010→9.03	100000→15.64	100001→16.66	0001001→20.09
0010→9.04	111110→15.64	011001→16.76	0110111→20.09
0100→9.04	000010→15.73	100110→16.76	1001000→20.09
1011→9.04	010000→15.73	000101→16.80	1110110→20.09
1101→9.04	101111→15.73	010111→16.80	0010010→20.11
0110→9.26	111101→15.73	101000→16.80	0100100→20.11
1001→9.26	010010→15.93	111010→16.80	1011011→20.11
0011→9.28	101101→15.93	001110→16.96	1101101→20.11
1100→9.28	010101→16.02	011100→16.96	0010101→20.15
00000→11.79	101010→16.02	100011→16.96	0101011→20.15
11111→11.79	010110→16.10	110001→16.96	1010100→20.15
00001→12.51	011010→16.10	001011→17.23	1101010→20.15
01111→12.51	100101→16.10	110100→17.23	0100101→20.16
10000→12.51	101001→16.10	000111→17.29	0101101→20.16
11110→12.51	001010→16.12	111000→17.29	1010010→20.16
00010→12.55	010100→16.12	0000000→18.03	1011010→20.16
01000→12.55	101011→16.12	1111111→18.03	0001010→20.22
10111→12.55	110101→16.12	0101010→18.68	0101000→20.22
11101→12.55	000100→16.15	1010101→18.68	1010111→20.22
00100→12.69	001000→16.15	0000001→18.92	1110101→20.22
11011→12.69	110111→16.15	0111111→18.92	0100001→20.26
01001→12.76	111011→16.15	1000000→18.92	0111101→20.26
01101→12.76	001001→16.38	1111110→18.92	1000010→20.26
10010→12.76	011011→16.38	0000010→19.10	1011110→20.26

number of printed 1s among all the produced strings, with a program-size complexity of 28 bits.

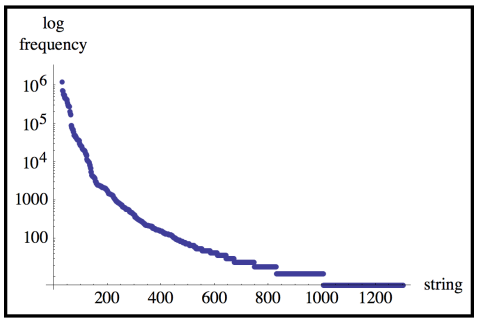

Figure 3.3: (4,2) output log-frequency plot, ordered from most to less frequent string, the slope is clearly exponential.

Same length string complexity

The classification table 10 allows to make a comparison of the structure of the strings related to their calculated complexity among all the strings of the same length extracted from $D(4)$.

Halting summary

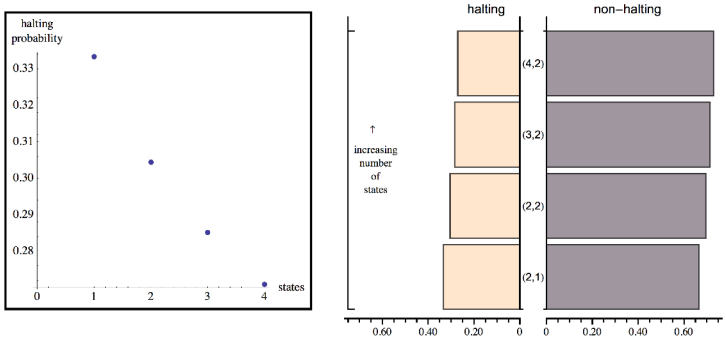

Figure 3.4: Graphs showing the halting probabilities among (n,2), $n < 5$. The list plot on the left shows the decreasing probability of the number of halting Turing machines while the paired bar chart on the right allows a visual comparison between both halting and non-halting machines side by side.

Table 3.10: Classification–from less to more random–for 7-bit strings extracted from $D(4)$.

0000000→18.03	1001000→20.09	0101001→20.42	0000111→20.99
1111111→18.03	1110110→20.09	0110101→20.42	0001111→20.99
0101010→18.68	0010010→20.11	1001010→20.42	1110000→20.99
1010101→18.68	0100100→20.11	1010110→20.42	1111000→20.99
0000001→18.92	1011011→20.11	0001100→20.48	0011110→21.00
0111111→18.92	1101101→20.11	0011000→20.48	0111100→21.00
1000000→18.92	0010101→20.15	1100111→20.48	1000011→21.00
1111110→18.92	0101011→20.15	1110011→20.48	1100001→21.00
0000010→19.10	1010100→20.15	0110110→20.55	0111110→21.03
0100000→19.10	1101010→20.15	1001001→20.55	1000001→21.03
1011111→19.10	0100101→20.16	0011010→20.63	0011001→21.06
1111101→19.10	0101101→20.16	0101100→20.63	0110011→21.06
0000100→19.38	1010010→20.16	1010011→20.63	1001100→21.06
0010000→19.38	1011010→20.16	1100101→20.63	1100110→21.06
1101111→19.38	0001010→20.22	0100010→20.68	0001110→21.08
1111011→19.38	0101000→20.22	1011101→20.68	0111000→21.08
0001000→19.45	1010111→20.22	0100110→20.77	1000111→21.08
1110111→19.45	1110101→20.22	0110010→20.77	1110001→21.08
0000110→19.64	0100001→20.26	1001101→20.77	0010011→21.10
0110000→19.64	0111101→20.26	1011001→20.77	0011011→21.10
1001111→19.64	1000010→20.26	0010110→20.81	1100100→21.10
1111001→19.64	1011110→20.26	0110100→20.81	1101100→21.10
0101110→19.68	0000101→20.29	1001011→20.81	0110001→21.13
0111010→19.68	0101111→20.29	1101001→20.81	0111001→21.13
1000101→19.68	1010000→20.29	0001101→20.87	1000110→21.13
1010001→19.68	1111010→20.29	0100111→20.87	1001110→21.13
0010001→20.04	0000011→20.38	1011000→20.87	0011100→21.19
0111011→20.04	0011111→20.38	1110010→20.87	1100011→21.19
1000100→20.04	1100000→20.38	0011101→20.93	0001011→21.57
1101110→20.04	1111100→20.38	0100011→20.93	0010111→21.57
0001001→20.09	0010100→20.39	1011100→20.93	1101000→21.57
0110111→20.09	1101011→20.39	1100010→20.93	1110100→21.57

In summary, among the (running over a tape filled with 0 only): 12, 3 044, 2 147 184 and 2 985 384 480 Turing machines in (n,2), $n < 5$, there were 36, 10 000, 7 529 536 and 11 019 960 576 that halted, that is slightly decreasing fractions of 0.333..., 0.3044, 0.2851 and 0.2709 respectively.

Full results can be found online at http://www.mathrix.org/experimentalAIT/

3.5.3 Runtimes investigation

Runtimes much longer than the lengths of their respective halting programs are rare and the empirical distribution approaches the *a priori* computable probability distribution on all possible runtimes predicted in [**?**]. As reported in [**?**] "long" runtimes are effectively rare. The longer it takes to halt, the less likely it is to stop.

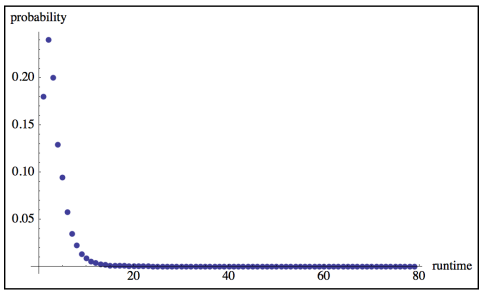

Figure 3.5: Runtimes distribution in (4,2).

Among the various miscellaneous facts from these results:

- All 1-bit strings were produced at $t = 1$.
- 2-bit strings were produced at all $2 < t < 14$ times.
- $t = 3$ was the time at which the first 2 bit strings of different lengths were produced ($n = 2$ and $n = 3$).
- Strings produced before 8 steps account for 49% of the strings produced by all (4,2) halting machines.
- There were 496 string groups produced by (4,2), that is strings that are not symmetric under reversion or complementation.
- There is a relation between t and n; no n-bit string is produced before $t < n$. This is obvious because a machine needs at least t steps to print t symbols.
- At every time t there was at least one string of length n for $1 < n < t$.

Table 3.11: Probability that a n-bit string among all $n < 10$ bit strings is produced at times $t < 8$.

	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$	
n=1	1.0	0	0	0	0	0	0	0
n=2	0	1.0	0.60	0.45	0.21	0.11	0.052	0.025
n=3	0	0	0.40	0.46	0.64	0.57	0.50	0.36
n=4	0	0	0	0.092	0.15	0.29	0.39	0.45
n=5	0	0	0	0	0	0.034	0.055	0.16
n=6	0	0	0	0	0	0	0	0.0098
n=7	0	0	0	0	0	0	0	0
n=8	0	0	0	0	0	0	0	0
n=9	0	0	0	0	0	0	0	0
n=10	0	0	0	0	0	0	0	0
Total	1	1	1	1	1	1	1	1

Table 3.12: Probability that a n-bit string with $n < 10$ is produced at time $t < 7$.

	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$	Total
n=1	0.20	0	0	0	0	0	0	0.20
n=2	0	0.14	0.046	0.016	0.0045	0.0012	0.00029	0.20
n=3	0	0	0.030	0.017	0.014	0.0063	0.0028	0.070
n=4	0	0	0	0.0034	0.0032	0.0031	0.0022	0.012
n=5	0	0	0	0	0	0.00037	0.00031	0.00069
n=6	0	0	0	0	0	0	0	0
n=7	0	0	0	0	0	0	0	0
n=8	0	0	0	0	0	0	0	0
n=9	0	0	0	0	0	0	0	0
n=10	0	0	0	0	0	0	0	0
Total	0.21	0.14	0.076	0.037	0.021	0.011	0.0057	

3.6 Discussion

Intuitively, one may be persuaded to assign a lower or higher algorithmic complexity to some strings when looking at tables 9 and 10, because they may seem simpler or more random than others of the same length. We think that very short strings may appear to be more or less random but may be as hard to produce as others of the same length, because Turing machines producing them may require the same quantity of resources to print them out and halt as they would with others of the same (very short) length.

For example, is 0101 more or less complex than 0011? Is 001 more or less complex than 010? The string 010 may seem simpler than 001 to us because we may picture it as part of a larger sequence of alternating bits, forgetting that such is not the case and that 010 actually was the result of a machine that produced it when entering into the halting state, using this extra state to somehow delimit the length of the string. No satisfactory argument may exist to say whether 010 is really more or less random than 001, other than actually running the machines and looking at their objective ranking according to the formalism and method described herein. The situation changes for larger strings, when an alternating string may in effect strongly suggest that it should be less random than other strings because a short description is possible in terms of the simple alternation of bits. Some strings may also assume their correct rank when the calculation is taken further, for example if we were able to compute $D(5)$.

On the other hand, it may seem odd that the program size complexity of a string of length l is systematically larger than l when l can be produced by a *print* function of length l+{the length of the print program}, and indeed one can interpret the results exactly in this way. The surplus can be interpreted as a constant product of a *print* phenomenon which is particularly significant for short strings. But since it is a constant, one can subtract it from all the strings. For example, subtracting 1 from all values brings the complexity results for the shortest strings to exactly their size, which is what one would expect from the values for algorithmic complexity. On the other hand, subtracting the constant preserves the relative order, even if larger strings continue having algorithmic complexity values larger than their lengths. What we provide herein, besides the numerical values, is a hierarchical structure from which one can tell whether a string is of greater, lesser or equal algorithmic complexity.

The *print program* assumes the implicit programming of the halting configuration. In C language, for example, this is delimited by the semicolon.

The fact then that a single bit string requires a 2 bit "program" may be interpreted as the additional information represented by the length of the string; the fact that a string is of length n is not the result of an arbitrary decision but it is encoded in the producing machine. In other words, the string not only carries the information of its n bits, but also of the delimitation of its length. This is different to, for example, approaching the algorithmic complexity by means of cellular automata–there being no encoded halting state, one has to manually stop the computation upon producing a string of a certain arbitrary length according to an arbitrary stopping time. This is a research program that we have explored before[25] and that we may analyze in further detail somewhere else.

It is important to point out that after the application of the coding theorem one often gets a non-integer value when calculating $C(s)$ from $m(s)$. Even though when interpreted as the size in bits of the program produced by a Turing machine it should be an integer value because the size of a program can only be given in an integer number of bits. The non-integer values are, however, useful to provide a finer structure providing information on the exact places in which strings have been ranked.

An open question is how much of the relative string order (hence the relative algorithmic probability and the relative algorithmic complexity) of $D(n)$ will be preserved when calculating $D(i)$ for larger Turing machine spaces such that $0 < n < i$. As reported here, $D(n)$ preserves most of the string orders of $D(n-1)$ for $1 < n < 5$. While each space (n,2) contains all (n-1,2) machines, the exponential increase in number of machines when adding states may easily produce strings such that the order of the previous distribution is changed. What the results presented here show, however, is that each new space of larger machines contributes in the same proportion to the number of strings produced in the smaller spaces, in such a way that they preserve much of the previous string order of the distributions of smaller spaces, as shown by calculating the Spearman coefficient indicating a very strong ranking correlation. In fact, some of the ranking variability between the distributions of spaces of machines with different numbers of states occurred later in the classification, likely due to the fact that the smaller spaces missed the production of some strings. For example, the first rank difference between $D(3)$ and $D(4)$ occurred in place 20, meaning that the string order in $D(3)$ was strictly preserved in $D(4)$ up to the top 20 strings sorted from higher to lower frequency. Moreover, one may ask whether the actual frequency values of the strings converge.

3.7 Concluding remarks

We have provided numerical tables with values of the algorithmic complexity for short strings. On the one hand, the calculation of $D(n)$ provides an empirical and *natural* distribution that does not depend on an additive constant and may be used in Bayesian contexts as a prior distribution. On the other hand, it might turn out to have valuable applications, especially because of the known issues explained in **??** related to the problem of approaching the complexity of short strings by means of compression algorithms often failing for short strings. This direct approach by way of algorithmic probability reduces the impact of the additive constant involved in the choice of computing model (or programming language), shedding light on the behavior of small Turing machines and tangibly grasping the concept of the algorithmic complexity of a string by exposing its structure on the basis of a complexity ranking.

To our knowledge this is the first time that numerical values of algorithmic complexity are provided and tabulated. We wanted to provide a table of values useful for several purposes, both to make conjectures about the continuation of the distribution–perhaps providing a statistical framework for a statistical calculation of the algorithmic complexity for longer strings–as well as to shed light on both the behavior of small Turing machines and the use of empirical approaches for practical applications of algorithmic complexity to the real-world.[7] The full tables are available online available at http://www.mathrix.org/experimentalAIT/

Bibliography

[1] A. H. Brady, *The Determination of the Value of Rado's Noncomputable Function* $\sum(k)$ *for Four-State Turing Machines.* Math. Comput. 40, 647-665, 1983.

[2] J-P. Delahaye and H. Zenil, On the Kolmogorov-Chaitin complexity for short sequences. in C.S. Calude (ed.) *Randomness and Complexity: From Leibniz to Chaitin.* World Scientific, 2007.

[3] G.J. Chaitin, *Algorithmic Information Theory.* Cambridge University Press, 1987.

7. For example, the authors of this paper have undertaken efforts to characterize and classify images by their complexity[26], for which these tables may be of great use.

[4] G.J. Chaitin, Computing the Busy Beaver function, in: T.M. Cover, B. Gopinath (Eds.), *Open Problems in Communication and Computation*, Springer-Verlag, Heidelberg, pp.108–112, 1987

[5] C.S. Calude, *Information and Randomness: An Algorithmic Perspective.* (Texts in Theoretical Computer Science. An EATCS Series), Springer; 2nd. edition, 2002.

[6] C.S. Calude and M.A. Stay, *Most programs stop quickly or never halt.* Advances in Applied Mathematics, 40 295–308, 2005.

[7] C.S. Calude, M.J. Dinneen, and C-K. Shu, *Computing a glimpse of randomness.* Experimental Mathematics, 11(2): 369-378, 2002.

[8] J. Hertel, *Computing the Uncomputable Rado Sigma Function: An Automated, Symbolic Induction Prover for Nonhalting Turing Machines.* The Mathematica Journal 11:2 2009.

[9] A. Holkner, *Acceleration Techniques for Busy Beaver Candidates.* Proceedings of the Second Australian Undergraduate Students' Computing Conference, 2004.

[10] W. Kirchherr, M. Li and P. Vitanyi, *The miraculous universal distribution.* Math. Intelligencer. 19(4), 7–15, 1997.

[11] A. N. Kolmogorov. *Three approaches to the quantitative definition of information.* Problems of Information and Transmission, 1(1): 1–7, 1965.

[12] A. Lempel and J. Ziv, *On the Complexity of Finite Sequences.* IEEE Trans. Inform. Theory, 22(1), pp. 75-81, 1976.

[13] L. Levin, *Laws of information conservation (non-growth) and aspects of the foundation of probability theory.* Problems In Form. Transmission 10, 206–210, 1974.

[14] L. Levin, *On a Concrete Method of Assigning Complexity Measures.* Doklady Akademii nauk SSSR, vol.18(3), pp. 727-731, 1977.

[15] L. Levin. *Universal Search Problems.* 9(3): 265-266, 1973 (c). (submitted: 1972, reported in talks: 1971). English translation in: B.A.Trakhtenbrot. *A Survey of Russian Approaches to Perebor (Brute-force Search) Algorithms.* Annals of the History of Computing 6(4): 384-400, 1984.

[16] M. Li, P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, 3rd. Revised edition, 2008.

[17] S. Lin and T. Rado, *Computer Studies of Turing Machine Problems.* J. ACM. 12, 196–212, 1965.

[18] R. Machlin, and Q. F. Stout, *The Complex Behavior of Simple Machines.* Physica 42D. 85-98, 1990.

[19] H. Marxen and J. Buntrock. *Attacking the Busy Beaver 5.* Bull EATCS 40, 247–251, 1990.

[20] T. Rado, *On noncomputable Functions.* Bell System Technical J. 41, 877–884, May 1962.

[21] R. Solomonoff, *A Preliminary Report on a General Theory of Inductive Inference.* (Revision of Report V-131), Contract AF 49(639)-376, Report ZTB–138, Zator Co., Cambridge, Mass., Nov, 1960.

[22] U. Speidel, *A note on the estimation of string complexity for short strings.* 7th International Conference on Information, Communications and Signal Processing (ICICS), 2009.

[23] A.M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem.* Proceedings of the London Mathematical Society. 2 42: 230–65, 1936, published in 1937.

[24] S. Wolfram, *A New Kind of Science.* Wolfram Media, 2002.

[25] H. Zenil and J.P. Delahaye, "On the Algorithmic Nature of the World," in Gordana Dodig-Crnkovic and Mark Burgin (eds), *Information and Computation.* World Scientific, 2010.

[26] H. Zenil, J.P. Delahaye and C. Gaucherel, *Information content characterization and classification of images by physical complexity.* arXiv: 1006.0051v1 [cs.CC].

[27] H. Zenil, *Busy Beaver.* from The Wolfram Demonstrations Project, http://demonstrations.wolfram.com/BusyBeaver/.

Troisième partie

Applications

Chapitre 4

Compression-Based Investigation of the Dynamical Properties of Cellular Automata and Other Systems

Published in H. Zenil, *Compression-Based Investigation of the Dynamical Properties of Cellular Automata and Other Systems*, Complex Systems, 19(1), pages 1-28, 2010.

4.1 Introduction

A method for studying the qualitative dynamical properties of abstract computing machines based on the approximation of their program-size complexity using a general lossless compression algorithm is presented. It is shown that the compression-based approach classifies cellular automata into clusters according to their heuristic behavior. These clusters show a correspondence with Wolfram's main classes of cellular automata behavior. A Gray code-based numbering scheme is developed for distinguishing initial

conditions. A compression-based method to estimate a characteristic exponent for detecting phase transitions and measuring the resiliency or sensitivity of a system to its initial conditions is also proposed. A conjecture regarding the capability of a system to reach computational universality related to the values of this phase transition coefficient is formulated. These ideas constitute a compression-based framework for investigating the dynamical properties of cellular automata and other systems.

4.2 Preliminaries

Previous investigations of the dynamical properties of cellular automata have involved compression in one form or another. In this paper, we take the direct approach, experimentally studying the relationship between properties of dynamics and their compression.

Cellular automata were first introduced by J. von Neumann [1] as a mathematical model for biological self-replication phenomena. They have since played a basic role in understanding and explaining various complex physical, social, chemical, and biological phenomena. Using extensive computer simulation S. Wolfram [2] classified cellular automata into four classes according to the qualitative behavior of their evolution. This classification has been further investigated and verified by G. Braga et al. [3], followed by more detailed verifications and investigations of classes 1, 2, and 3 in [4, 5].

Other formal approaches to the problem of classifying cellular automata have also been attempted, with some success. Of these, some are based on the structure of attractors or other topological classifications [6, 7], others use probabilistic approaches [8] or involve looking at whether a cellular automaton falls into some chaotic attractor or an undecidable class [9–11], while yet others use the idea of approaching the algorithmic or program-size complexity (K) of the rule table of a cellular automaton [12]. It has also been shown that rules that in certain conditions belong to one class may belong to another when starting from a different set up. This has been the case of rule 40, simple and therefore in class 1 when starting from a 0-finite configuration but chaotic [13] when starting from certain random configurations.

Compression-based mathematical characterizations and techniques for classifying and clustering have been suggested and successfully developed in areas as diverse as languages, literature, genomics, music, and astronomy. A good introduction can be found in [14]. Compression is a powerful tool for pattern recognition and has often been used for classification and

clustering. Lempel–Ziv (LZ)-like data compressors have been proven to be universally optimal and are therefore good candidates as approximators of the program-size complexity of strings.

The program-size complexity [15] $K_u(s)$ of a string s with respect to a universal Turing machine U is defined as the binary length of the shortest program p that produces as output the string s. Or, as a *Mathematica* expression:

$K_u(s) = \{min(Length[p]), U(p) = s\}$

However, a drawback of K is that it is an uncomputable function. In general, the only way to approach K is by compressibility methods. Essentially, the program-size complexity of a string is the ultimate compressed version of that string.

As an attempt to capture and systematically study the behavior of abstract machines, our experimental approach consists of calculating the program-size complexity of the output of the evolution of a cellular automaton. This is done following methods of extended computation, enumerating, and exhaustively running the systems as suggested in [2].

A cellular automaton is a collection of cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are applied iteratively for as many time steps as desired. The number of colors (or distinct states) k of a cellular automaton is a non-negative integer. In addition to the grid on which a cellular automaton lives and the colors its cells may assume, the neighborhood over which cells affect one another must also be specified. The simplest choice is a nearest-neighbor rule, in which only cells directly adjacent to a given cell may be affected at each time step. The simplest type of cellular automaton is then a binary, nearest-neighbor, one-dimensional automaton (called *elementary* by Wolfram). There are 256 such automata, each of which can be indexed by a unique binary number whose decimal representation is known as the rule for the particular automaton.

Regardless of the apparent simplicity of their formal description, cellular automata are capable of displaying a wide range of interesting and different dynamical properties as thoroughly investigated by Wolfram in [2]. The problem of classification is a central topic in cellular automata theory.

Wolfram identifies and classifies cellular automata (and other discrete systems) as displaying these four different classes of behavior.

1. A fixed, homogeneous state is eventually reached (e.g., rules 0, 8, 136).

2. A pattern consisting of separated periodic regions is produced (e.g., rules 4, 37, 56, 73).
3. A chaotic, aperiodic pattern is produced (e.g., rules 18, 45, 146).
4. Complex, localized structures are generated (e.g., rule 110).

4.3 Compression-based classification

The method consists of compressing the evolution of a cellular automaton up to a certain number of steps. The *Mathematica* function Compress [16] gives a compressed representation of an expression as a string. It uses a C language implementation of a "deflate" compliant compressor and decompressor available within the zlib package. The deflate lossless compression algorithm, independent of CPU type, operating system, file system, and character set compresses data using a combination of the LZ algorithm and Huffman coding [17–19], with efficiency comparable to the best currently available general-purpose compression methods as described in RFC 1951 [20] called the Lempel-Ziv-Welch (LZW) algorithm. The same algorithm is the basis of the widely used gzip data compression software. Data compression is generally achieved through two steps:

- The matching and replacement of duplicate strings with pointers.
- Replacing symbols with new, weighted symbols based on frequency of use.

4.3.1 Compression-based classification of elementary cellular automata from simplest initial conditions

The difference in length between the compressed and uncompressed forms of the output of a cellular automaton is a good approximation of its program-size complexity. In most cases, the length of the compressed form levels off, indicating that the cellular automaton output is repetitive and can easily be described. However, in cases like rules 30, 45, 110, or 73 the length of the compressed form grows rapidly, corresponding to the apparent randomness and lack of structure in the display.

Classification parameters

There are two main parameters that play a role when classifying cellular automata: the initial configuration and the number of steps. Classifying cellular automata can begin by starting all with a single black cell. Some of them, such as rule 30, will immediately show their full richness in dynamical terms, while others might produce very different behavior when starting with another initial configuration. Both types might produce different classifications. We first explore the case of starting with a single black cell and then proceed to consider the other case for detecting phase transitions.

An illustration of the evolution of rules 95, 82, 50, and 30 is shown in Figure 1, together with the compressed and uncompressed lengths they produce, each starting from a single black cell moving through time (number of steps).

As shown in Figure 1, the compressed lengths of simple cellular automata do not approach the uncompressed lengths and stay flat or grow linearly, while the length of the compressed form approaches the length of the uncompressed form for rules such as 30.

Cellular automata can be classified by sorting their compressed lengths as an approximation to their program-size complexity. In Figure 2, c is the compressed length of the evolution of the cellular automaton up to the first 200 steps (although the pictures only show the first 60).

Early in 2007 I wrote a program using *Mathematica* to illustrate the basic idea of the compressibility method for classifying cellular automata. The program (called a Demonstration) was submitted to the Wolfram Demonstrations Project and published under the title "Cellular Automaton Compressibility" [21]. Later in 2007, inspired by this Demonstration and under my mentorship, Joe Bolte from Wolfram Research, Inc. developed a project under the title "Automatic Ranking and Sorting of Cellular Automaton Complexity" at the NKS Summer School held at the University of Vermont (for more information see http://www.wolframscience.com/summerschool/2007/participants/bolte.html). In 2009, also under my mentorship, Chiara Basile from the University of Bologna would further develop the project at the NKS Summer School held at the ISTI-CNR in Pisa, Italy, under the title "Exploring CA Rule Spaces by Means of Data Compression." The project was enriched by Basile's own prior research, particularly on feeding the compression algorithm with sequences following different directions (rows, columns, and space-filling curves), thus helping speed up the pattern detection process (for more information see http://www.wolframscience.com/summerschool/2009/participants/basile.html).

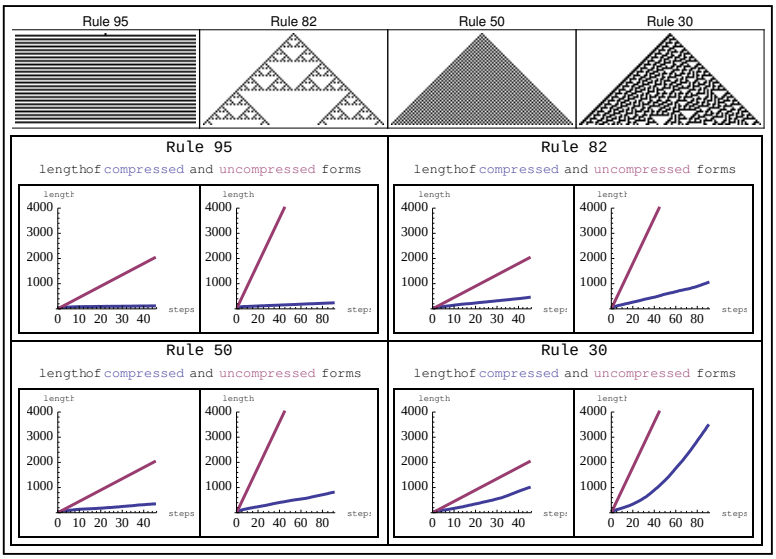

Figure 4.1: Evolution of rules 95, 82, 50, and 30 together with the compressed (dashed line) and uncompressed (solid line) lengths.

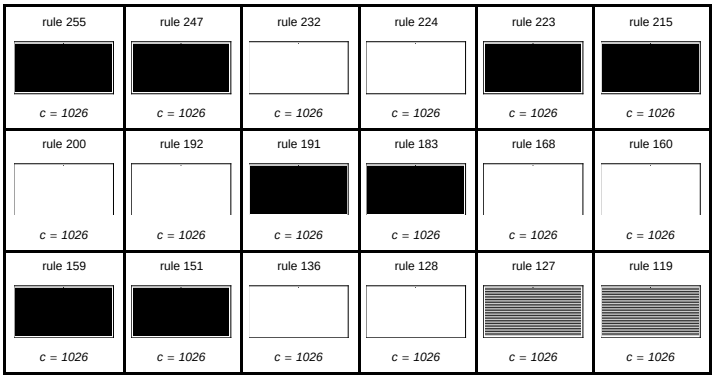

Figure 4.2: Complete compression-based classification of the elementary cellular automata (*continues*).

4.4 Compression-based clustering

4.4.1 2-clusters plot

By finding neighboring clusters of compressed lengths cellular automata can be grouped by their program-size complexity. Treating pairs of elements as being less similar when their distances are larger using an Euclidean distance function, a 2-clusters plot was able to separate cellular automata that

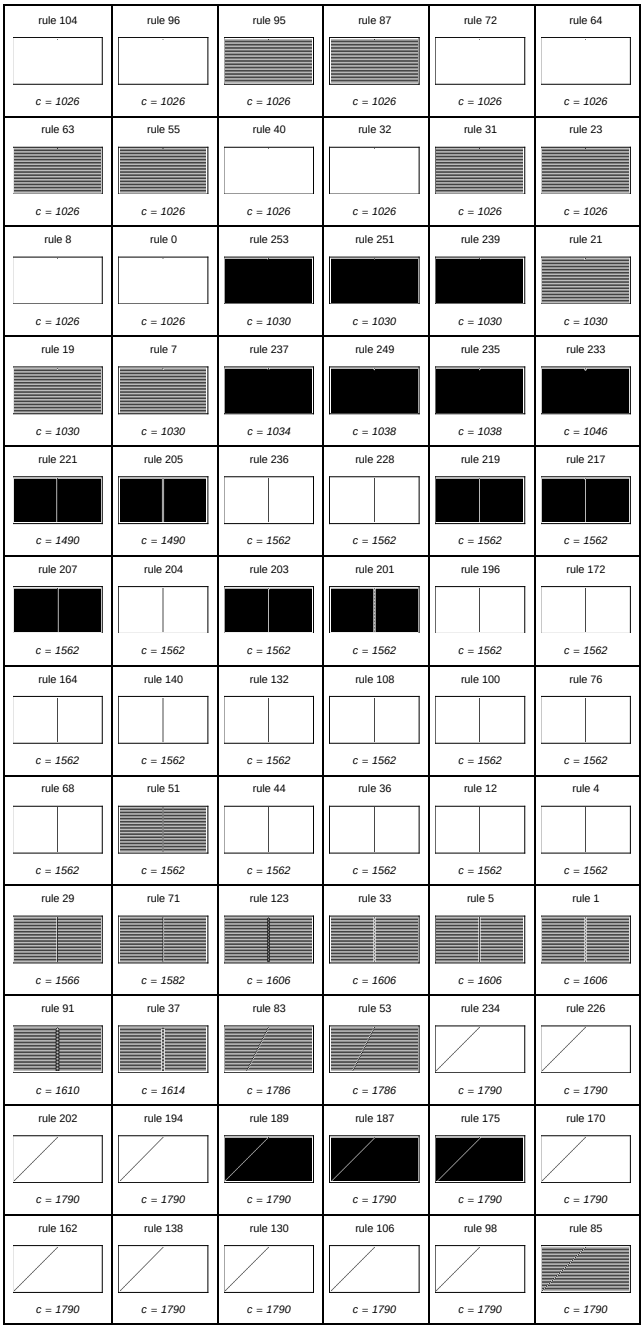

Figure 4.3: (*continued*).

clearly fall into Wolfram's classes 3 and 4 from the rest, dividing complex and random-looking cellular automata from trivial and nested ones as shown in Figures 3 through 5.

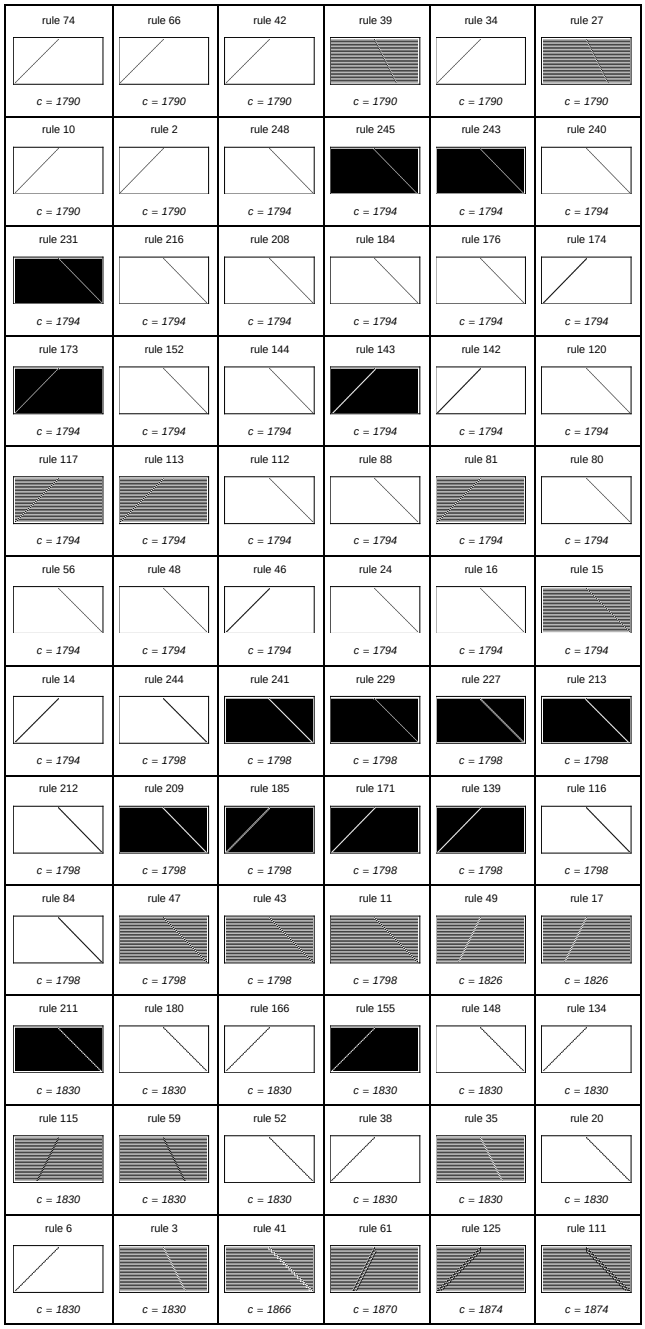

Figure 4.4: (*continued*).

A second application of the clustering algorithm splits the original classes 3 and 4 into clusters linking automata by their qualitative properties as shown in Figure 6.

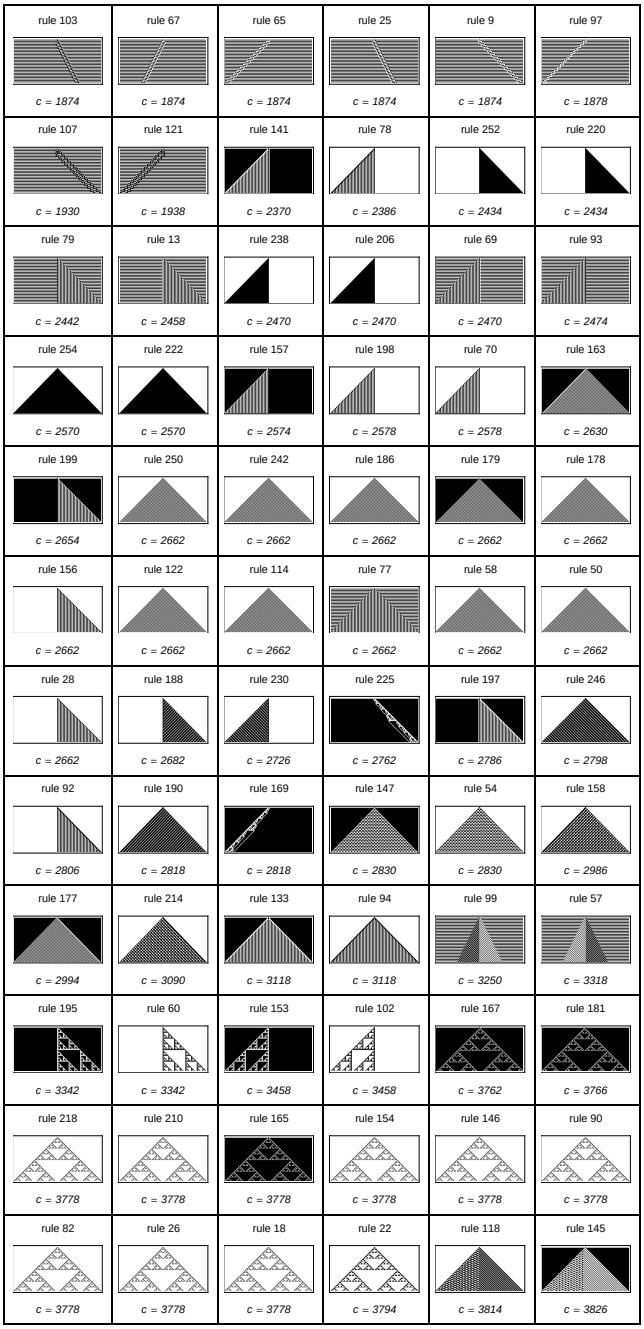

Figure 4.5: (*continued*).

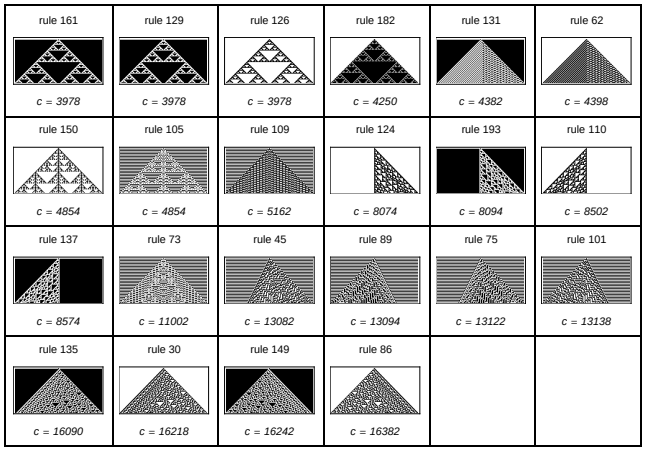

Figure 4.6: (*continued*).

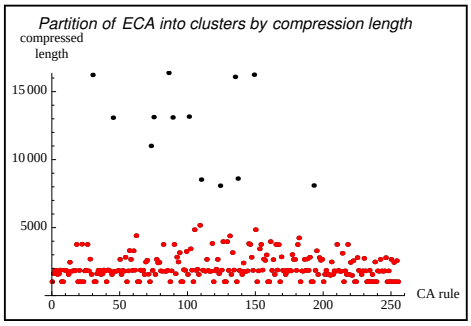

Figure 4.7: Partitioning elementary cellular automata into clusters by compression length.

4.4.2 Compression-based classification of larger spaces of cellular automata and other abstract machines

3-color nearest-neighbor cellular automata

By following the same technique, we were able to identify 3-color nearest-neighbor cellular automata in classes 3 and 4 as shown in Figure 7.

2-state 3-color Turing machines

The exploration of Turing machines is considerably more difficult for three main reasons.

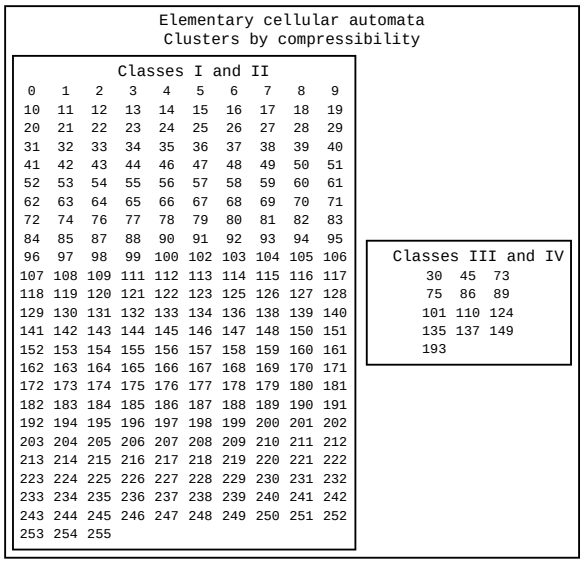

Figure 4.8: Elementary cellular automata clusters by compressibility.

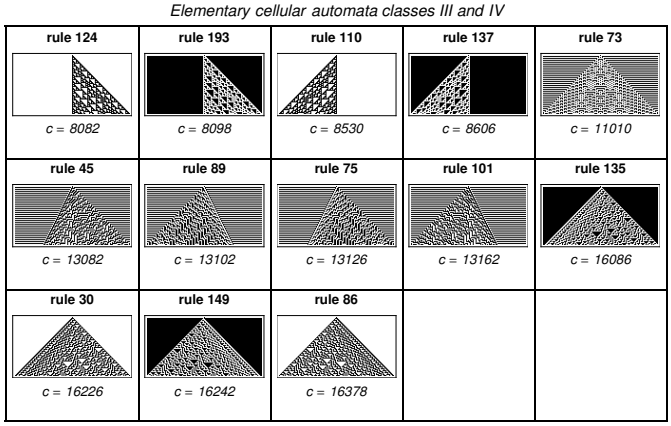

Figure 4.9: Elementary cellular automata classes 3 and 4.

1. The spaces of Turing machines for the shortest states and colors are much larger than the shortest spaces of cellular automata.

2. Turing machines with nontrivial dynamical properties are very rare compared to the size of the space defined by the number of states and colors, and therefore larger samples are necessary.

3. Turing machines evolve much more slowly than cellular automata, so longer runtimes are also necessary.

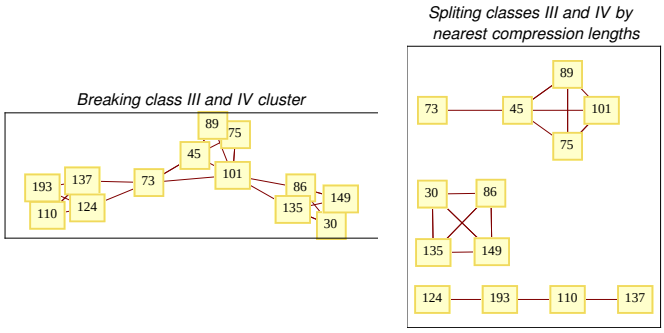

Figure 4.10: (a) Breaking class 3 and 4 clusters. (b) Splitting classes 3 and 4 by nearest compression lengths.

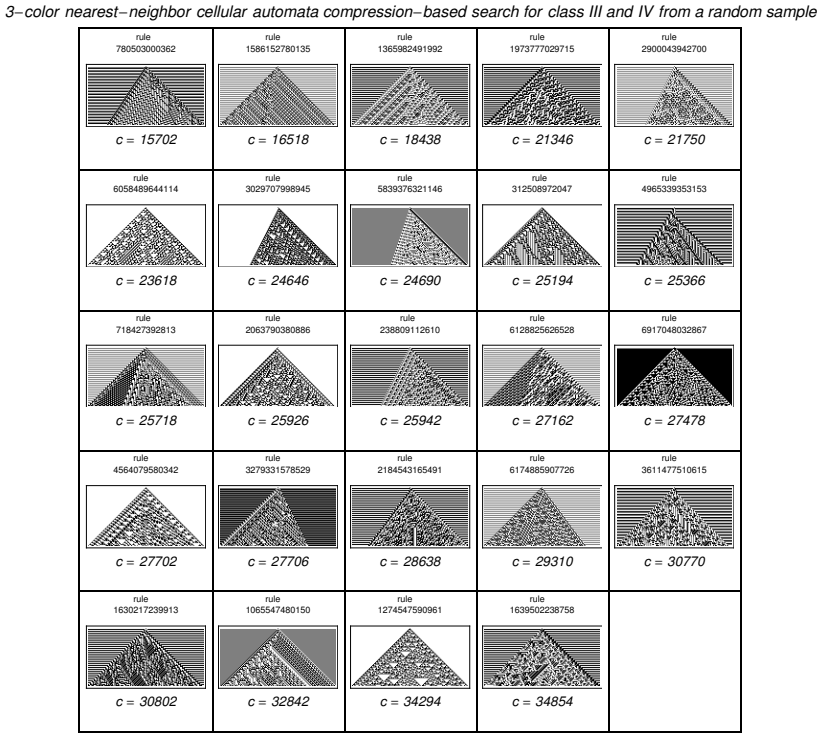

Figure 4.11: 3-color nearest-neighbor cellular automata compression-based search for classes 3 and 4 from a random sample.

The best compression-based results for identifying nontrivial Turing machines were obtained by applying the technique to the number of states a Turing machine is able to reach after a certain number of steps rather than to the output itself, unlike for cellular automata. The complexity of a Turing machine is deeply determined by the number of states the machine is

capable of reaching from an initial configuration, and by looking at its complexity the technique distinguished the nontrivial machines from the most trivial. Figure 8 shows a sample of Turing machines found by applying the compression-based method.

4.5 Compression-based phase transition detection

A phase transition can be defined as a discontinuous change in the dynamical behavior of a system when a parameter associated with the system, such as its initial configuration, is varied.

It is conventional to distinguish two kinds of phase transitions, often called first- and higher-order. As described in [2], one feature of first-order transitions is that as soon as the transition is passed, the whole system always switches completely from one state to the other. However, higher-order transitions are gradual or recurrent. On one side of the transition a system is typically completely ordered or disordered. But when the transition is passed, the system does not change from then on to either one or another state. Instead, its order increases or decreases more or less gradually or recurrently as the parameter is varied. Typically the presence of order is signaled by the breaking of some kind of symmetry; for example, two rules explored in this section (rules 22 and 109) were found to be highly disturbed with recurrent phase transitions due to a symmetry breaking when starting with certain initial configurations.

4.5.1 Initial configuration numbering scheme

Ideally, one should feed a system with a natural sequence of initial configurations of gradually increasing complexity. Doing so assures that qualitative changes in the evolution of the system are not attributable to discontinuities in its set of initial conditions.

The reflected binary code, also known as the "Gros–Gray code" or simply the "Gray code" (after Louis Gros and Frank Gray), is a binary numeral system where two successive values differ by only one bit. To explore the qualitative behavior of a cellular automaton when starting from different initial configurations, the optimal method is to follow a Gros–Gray encoding enumeration in order to avoid any undesirable "jumps" attributable to the system's having been fed with discontinuous initial configurations. By

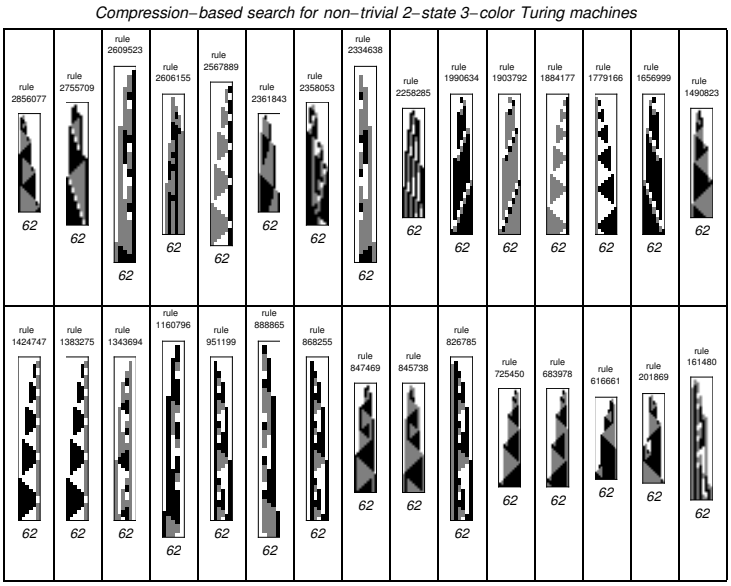

Figure 4.12: Compression-based search for nontrivial 2-state 3-color Turing machines.

following the Gros–Gray code, an optimal numbering scheme was devised so that two consecutive initial conditions differ only by the simplest change (one bit).

GrosGrayCodeDerivate and GrosGrayCodeIntegrate implement the methods described in [22].

$GrosGrayCodeDerivate[n\_Integer] := Prepend[Mod[\#[[1]] + \#[[2]], 2]\&/@ Partition[\#, 2, 1], \#[[1]]]\&@IntegerDigits[n, 2]$

$GrosGrayCodeIntegrate[l\_List] := FromDigits[Mod[\#, 2]\&/@Accumulate[l], 2]$

The function InitialConfiguration implements the optimal numbering scheme of initial conditions for cellular automata based on the Gros–Gray code, minimizing the Damerau–Levenshtein distance.

GrosGrayCodeIntegrate is the reverse function of GrosGray˙˙.CodeDerivate. It retrieves the element number of an element in Gros–Gray's code, that is, the composition of GrosGrayCodeDerivate and GrosGrayCodeIntegrate is the identity function.

$Table[GrosGrayCodeIntegrate[GrosGrayCodeDerivate[n]], \{n, 0, 10\}]$

$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

The Damerau–Levenshtein distance between two vectors u and v gives the number of one-element deletions, insertions, substitutions, and transpositions

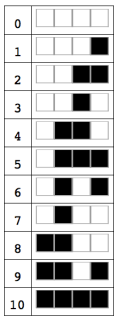

Figure 4.13: First 11 elements of the Gros–Gray code.

required to transform u into v. It can be verified that the distance between any two adjacent elements in the Gros–Gray code is always 1.

DamerauLevenshteinDistance[#[[1]], #[[2]]]&/@
Partition[*Table*[*GrosGrayCodeDerivate*[n], {n, 0, 10}], 2, 1]

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}

The simplest, not completely trivial, initial configuration of a cellular automaton is the typical single black cell that can be denoted (as in *Mathematica*) by {{1}, 0}, meaning a single black cell (1) on a background of whites (0). Preserving an "empty" background leaves the region that must be varied consisting only of the nonwhite portion of the initial configuration. However, when surrounded with zeroes, initial configurations may be the same for cellular automata. For example, the initial configuration {0, 1, 0} is exactly the same as {1} because the cellular automaton background is already filled with zeroes. Therefore, valid different initial configurations for cellular automata should always be wrapped in 1s.

InitialCondition[*n_Integer*]:=*If*[n===0, {*Last*[#]}, #]&@
  Append[*GrosGrayCodeDerivate*[n], 1]

InitialConditionNumber is the reverse function for retrieving the number of an initial configuration given an initial configuration according to the numbering scheme devised herein.

InitialConditionNumber[*l_List*]:=
GrosGrayCodeIntegrate[*Most*[l]]

For example, the thirty-second initial condition is

Figure 4.14: *InitialConditionNumber*[*InitialCondition*[32]]

32

An interesting example is the elementary cellular automaton rule 22, which behaves as a fractal in Wolfram's class 2 for some segment of initial configurations, followed by phase transitions of more disordered evolutions of the type of class 3.

4.5.2 Phase transitions

Two one-dimensional elementary cellular automata that show discrete changes in behavior when the properties of their initial conditions are continuously changed are shown in Figures 11 and 12.

Data points are joined for clarity only. It can be seen that up to the initial configuration number 20 there are clear spikes at initial configurations 8, 14, 17, and 20 indicating four abrupt phase transitions.

For clarity, the background of the evolution of rule 109 in Figure 12 was cleaned up. Clear phase transitions are detected at initial configuration numbers 2, 3, 11, and 13, together with weaker behavior changes at initial configuration numbers 15, 16, and 20 that only occur on one side of the cellular automaton and therefore show spikes firing at half the length.

Comparison of the sequences of the compressed lengths of six different elementary cellular automata following the initial configuration numbering scheme up to the first $2^7 = 128$ initial configurations up to 150 steps each is shown in Figure 13.

The differences between the compressed versions provide information on the changes in behavior up to a given number of steps of a system starting from different initial conditions. The normalization divided by the number of steps provides the necessary stability to keep the increase of complexity on account of the increase of size due to longer runtimes out of the main equation. In other words, the program-size complexity accumulated due to longer runtimes is subtracted in time from the approximated program-size complexity of the system itself.

The method given can also be used to precompute the initial configurations of a cellular automaton space conducting the search for interesting behaviors and speeding up the study of qualitative dynamical properties. For example, interesting initial configurations to look at for rules 22 and 109 are those detected in Figure 13 showing clear phase transitions.

One open question is whether there are first-order phase transitions (when following a "natural" initial condition enumeration) in elementary cellular

automata. Our method was only capable of detecting higher-order phase transitions up to the steps and initial conditions explored herein.

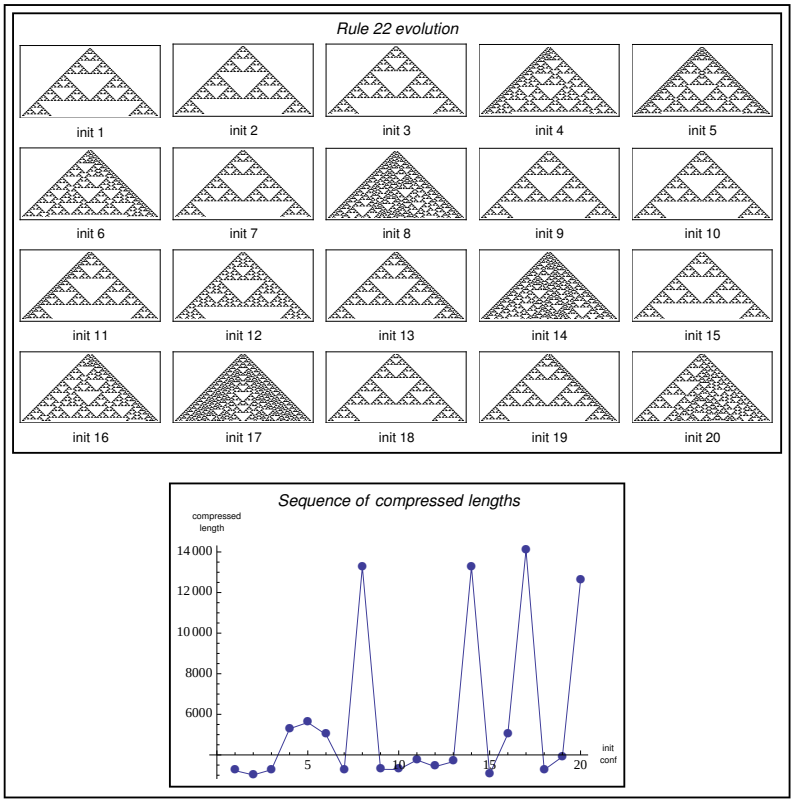

Figure 4.15: Rule 22 phase transition.

4.6 Compression-based numerical computation of a characteristic exponent

A fundamental property of chaotic behavior is the sensitivity to small changes in the initial conditions. Lyapunov characteristic exponents quantify this qualitative behavior by measuring the mean rate of divergence of initially neighboring trajectories. A characteristic exponent as a measure usually has the advantage of keeping systems with no significant phase transitions close to a constant value, while those with significant phase transitions are distinguished by a linear growth that characterizes instability in the system.

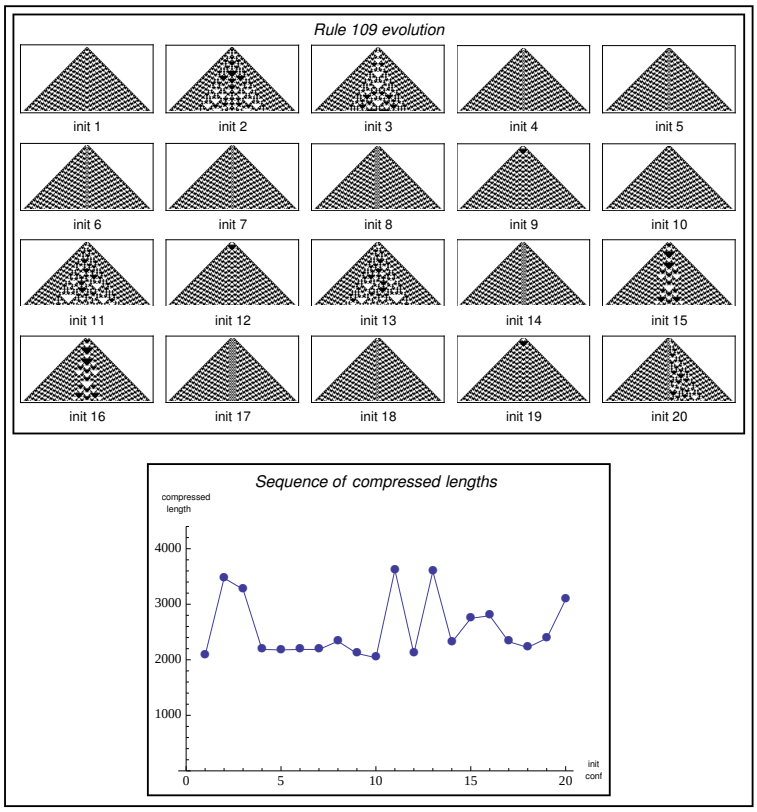

Figure 4.16: Rule 109 phase transition.

Whether a system has a phase transition is an undecidable property of systems in general. However, the characteristic exponent is an effective method of calculation, even with no prior knowledge of the generating function of the system.

The technique described herein consists of comparing the mean of the divergence in time of the compressed lengths of the output of a system running over a sequence of small changes to the initial conditions over small intervals of time. The procedure yields a sequence of values normalized by the runtime and the derivative of the function that best fits the sequence. Just as with Wolfram's method described in [2] for the calculation of the Lyapunov exponents of a cellular automaton, the divergence in time is measured by the differences in space-time of the patterns produced by the system. But unlike the calculation of Lyapunov exponents, this will be done by measuring the distance between the compressed regions of the evolution of a cellular automaton when starting from different initial configurations. After normalization, we will be able to evaluate a stable characteristic exponent and

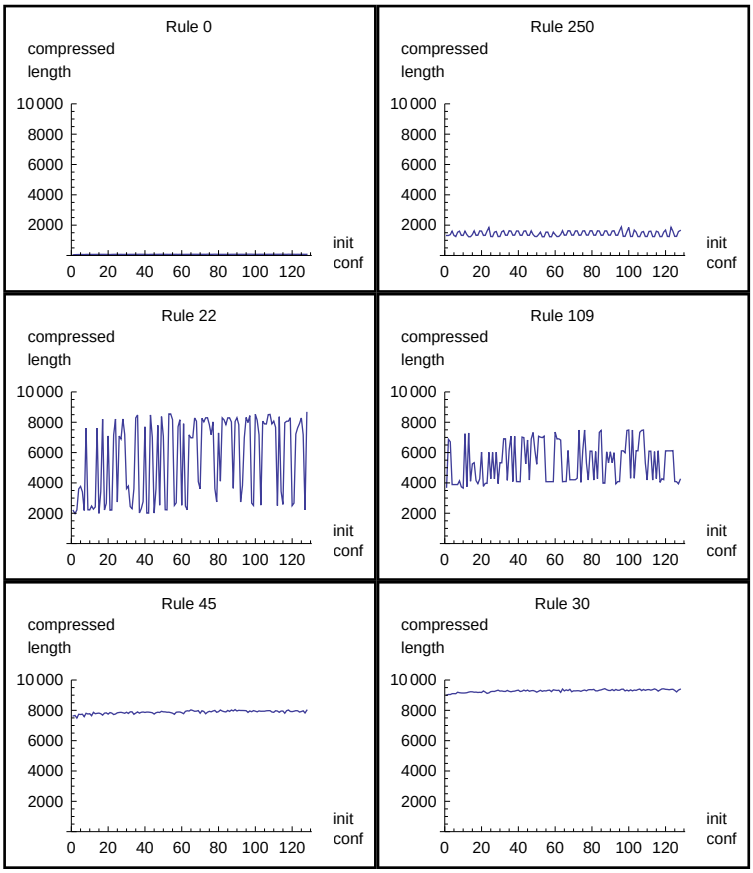

Figure 4.17: Sequence of compressed lengths for six elementary cellular automata.

therefore characterize its degree of sensitivity.

We want to examine the relative behavior of region evolutions when starting from adjacent initial configurations. Since the systems for which we are introducing this method are discrete, the regular parameter separating the initial conditions in a continuous system when calculating the Lyapunov exponent of a system can be replaced by the immediate successor of an initial condition following an optimal enumeration like the one described in Section 4 based on the Gros–Gray code.

Let the characteristic exponent c_n^t be defined as the mean of the absolute values of the differences of the compressed lengths of the outputs of the system M running over the initial segment of initial conditions i_j with $j = \{1, \ldots, n\}$ following the numbering scheme devised earlier and running for t steps, as follows:

$$c_n^t = \frac{|C(M_t(i_1))-C(M_t(i_2))|+\cdots+|C(M_t(i_{n-1}))-C(M_t(i_n))|}{t(n-1)}$$

The division by t acts as a normalization parameter in order to keep the runtime of the different values of c_n^t as independent as possible among the systems. However, as already noted, normalization can also be achieved by dividing by the "volume" of the region (the space-time diagram) generated by a system (in the case of a one-dimensional cellular automaton the area, i.e., the number of affected cells—usually the characteristic cone). The mean of the absolute values can also be replaced by the maximum of the absolute values in order to maximize the differences depending on the type of dynamical features being intensified.

Let us define a phase transition sequence as the sequence of characteristic exponents for a system M running for longer runtimes.

S_c	$f(S_c)$
$\{3.0, 5.2, 7.5, 9.9, 12., 15., 17., 20., 22., 24., \ldots\}$	$0.0916163 + 2.4603x$
$\{2.6, 2.3, 3.6, 5.0, 6.7, 8.6, 9.6, 11., 12., 14., \ldots\}$	$-0.0956435 + 1.39588x$
$\{1.5, 2.3, 3.2, 4.6, 5.8, 7.1, 8.7, 10., 11., 13., \ldots\}$	$-0.35579 + 1.28849x$
$\{4.0, 6.3, 8.6, 11., 13., 15., 17., 20., 22., 24., \ldots\}$	$1.48149 + 2.30786x$
$\{2.5, 3.3, 4.4, 5.6, 6.6, 7.3, 7.5, 8.0, 8.7, 9.1, \ldots\}$	$3.52132 + 0.492722x$
$\{3.8, 4.3, 4.7, 5.0, 5.4, 5.6, 5.9, 6.4, 6.7, 7.2, \ldots\}$	$3.86794 + 0.296409x$
$\{2.4, 3.7, 4.6, 5.4, 5.7, 6.0, 6.2, 6.4, 6.6, 6.7, \ldots\}$	$4.2508 + 0.184981x$
$\{1.8, 1.8, 2.1, 2.8, 3.4, 3.8, 4.1, 4.4, 4.7, 4.8, \ldots\}$	$1.83839 + 0.270672x$
$\{1.9, 3.0, 2.8, 3.3, 3.7, 4.0, 4.4, 4.7, 4.9, 5.0, \ldots\}$	$2.57937 + 0.207134x$
$\{2.0, 3.1, 3.1, 3.7, 4.3, 4.6, 4.8, 5.1, 5.4, 5.4, \ldots\}$	$2.89698 + 0.218607x$
$\{0.61, 0.45, 0.38, 0.39, 0.28, 0.30, 0.24, 0.28, 0.35, 0.43, \ldots\}$	$0.41144 - 0.00298547x$
$\{0.35, 0.31, 0.40, 0.42, 0.56, 0.62, 0.72, 0.90, 1.2, 1.4, \ldots\}$	$-0.751501 + 0.268561x$
$\{0.48, 0.41, 0.29, 0.37, 0.42, 0.42, 0.47, 0.51, 0.52, 0.55, \ldots\}$	$0.302027 + 0.0263495x$
$\{0.087, 0.057, 0.038, 0.036, 0.027, 0.028, 0.024, 0.019, 0.017, 0.021, \ldots\}$	$0.0527182 - 0.0028416x$

Table 1. Regression analysis.

The general rule for $t = 200$ and $n = 40$ is that if the characteristic exponent c_n^t is greater than 1 for large enough values of n and t, then c_n^t has a phase transition. Otherwise it does not. Table 1 shows the calculation of the characteristic exponents of some elementary cellular automata.

4.6.1 Regression analysis

Let $S_c = S(c_n^t)$ for a fixed n and t. The line that better fits the growth of a sequence S_c can be found by calculating the linear combination that minimizes the sum of the squares of the deviations of the elements. Let

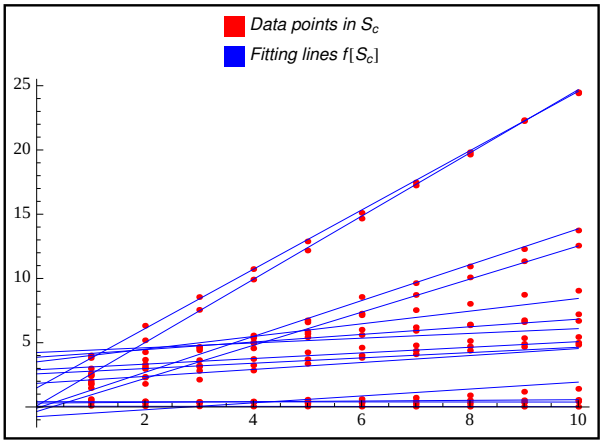

Figure 4.18: Regression analysis: fitting the point into a linear equation.

$f(S_c)$ denote the line that fits the sequence S_c by finding the least-squares as shown in Figure 15.

The derivatives of a phase transition function are therefore stable indicators of the degree of the qualitative change in behavior of the systems. The larger the derivative, the larger the significance. Let C denote the transition coefficient defined as $C = f'(S_c)$. Table 2 illustrates the calculated transition coefficients for a few elementary cellular automata rules.

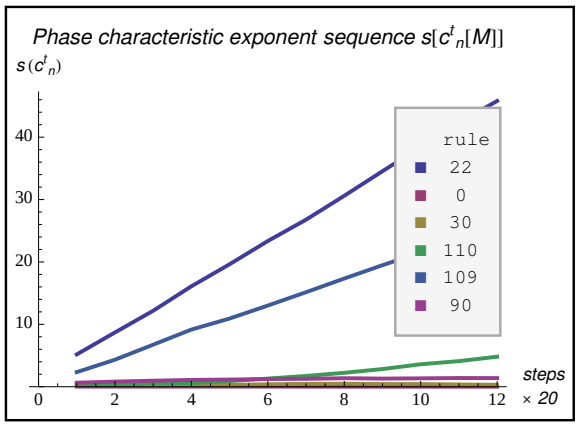

Figure 4.19: Phase characteristic exponent sequence $s(c_n^t(M))$.

Interesting initial conditions

After calculating the transition coefficient, we can calculate the first 10 most interesting initial conditions for elementary cellular automata with transition coefficients greater than 1. Those listed in Table 3 were calculated up to 600 steps in blocks of 50.

ECA Rule (r)	$C\,(\mathbf{ECA}_r)$
22	2.5
151	2.3
109	1.4
73	1.3
133	0.49
183	0.30
54	0.27
110	0.27
97	0.22
41	0.21
147	0.18
45	0.026
1	-0.0028
30	-0.0030

Table 2. Phase transition coefficient.

ECA Rule	**Initial Configuration Number**
151	$\{17, 18, 20, 22, 26, 34, 37, 41, 44, 46\}$
22	$\{8, 14, 17, 20, 23, 24, 26, 27, 28, 29\}$
73	$\{10, 12, 15, 16, 21, 24, 28, 30, 43, 45\}$
109	$\{2, 3, 11, 13, 20, 24, 26, 28, 32, 33\}$
133	$\{4, 6, 8, 10, 14, 16, 18, 19, 22, 25\}$
94	$\{10, 16, 20, 23, 24, 26, 28, 29, 30, 32\}$

Table 3. Elementary cellular automata that are the most sensitive to initial configurations.

4.6.2 Phase transition classification

The coefficient C has positive values if the system is sensitive to the initial configurations. The larger the positive values, the more sensitive the system is to initial configurations. C has negative values or is close to 0 if it is

highly homogeneous. Irregular behavior yields nonlinear growth, leading to a positive exponent.

For elementary cellular automata, it was found that $n = 40$ and $t = 200$ were runtime values large enough to detect and distinguish cellular automata having clear phase transitions. It is also the case that systems showing no quick phase transition have an asymptotic probability 1 of having a transition at a later time. In other words, a system with a phase transition either has the transition very early in time or is unlikely to ever have one later, as can be theoretically predicted from an algorithmic theoretical argument. (A phase transition is undecidable and can be seen as a reachability problem equivalent to the halting problem, hence a system powerful enough to halt when reaching a phase transition, as calculated earlier, has an [effective] density zero[23].) The same transition coefficient can also be seen as a homogeneity measure. At the right granular level, randomness shares informational properties with trivial systems. Like a trivial system, a random system is incapable of transmitting or carrying out information. The characteristic exponent relates these two behaviors in an interesting way since the granularity of a random system for a runtime large enough is close to the dynamical state of being in a stable configuration according to this measure. One can see that rule 110 is better classified, certainly because it has some structure and is less homogeneous in time, unlike rule 30 and of course rule 1. Rule 30, like rule 1, changes its compressed output from one step to the other at a lower rate or not at all. While the top of the classification and the gap between them are more significant because they show a qualitative change in their evolution, the bottom is classified by its lack of changes. In other words, while rules like 22 and 151 exhibit more changes when starting from different initial classifications, rules such as 30 and 1 always look alike.

The clusters formed (a different cluster per row) for a few selected cellular automata rules starting from random initial conditions are shown in Figure 16.

The clusters shown in Figure 16 are clearly classifying this small selection of cellular automata by the presence of phase transitions (sudden structures). This is also a measure of homogeneity.

A measure of homogeneity for classifying elementary cellular automata according to their transition coefficients can be calculated. The top 24 and bottom 22 cellular automata up to 600 steps in blocks of 50 for the first 500 initial conditions followed by their transition coefficients sorted from larger to smaller values are shown in Figures 17 and 18. The complete table of transition coefficients is available at http://www.algorithmicnature.org.

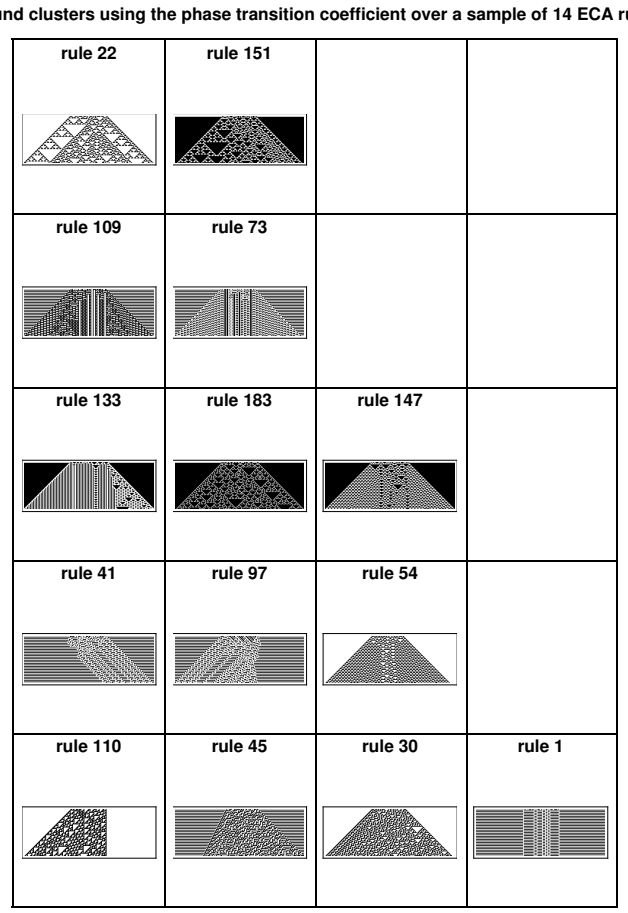

Figure 4.20: Clusters found using the phase transition coefficient over a sample of 14 rules.

Since a random system would be expected to produce a homogeneous stream with no distinguishable patterns as incapable of carrying or transmitting any information, both the simplest and the random systems were classified together at the end of the figure.

Conjecture relating universality and the phase transition coefficient

Based on this study, we conjecture that a system will be capable of universal computation if it has a large transition coefficient, at least larger than zero, say. The inverse, however, should not hold, because having a large transition coefficient by no means implies that the system will behave with the freedom required of a universal system if it is to emulate any possible com-

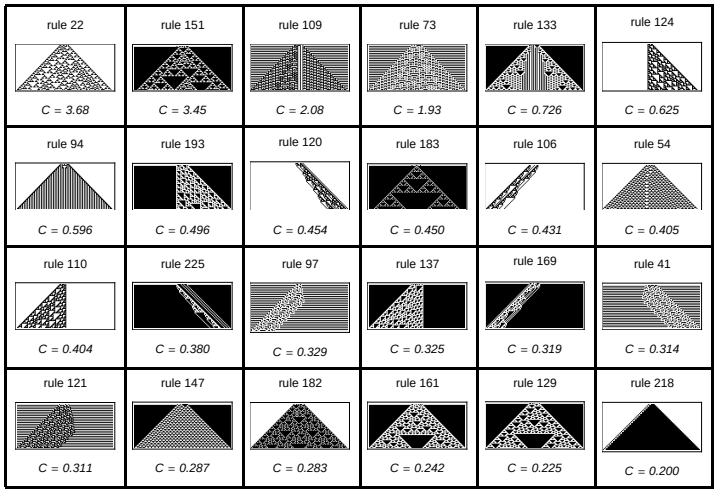

Figure 4.21: Phase transition coefficient classification (bottom 22) of elementary cellular automata (picture displaying a random initial configuration).

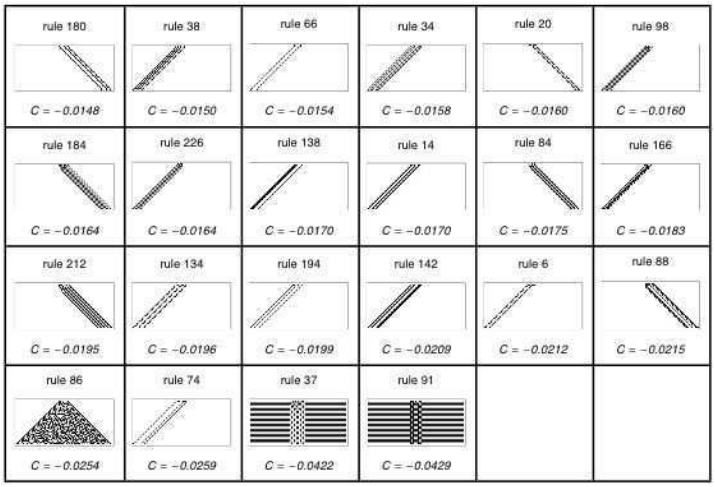

Figure 4.22: Phase transition coefficient classification (bottom 20) of elementary cellular automata (picture displaying a random initial configuration).

putation (a case in point may be rule 22, which, despite having the largest transition coefficient, may not be versatile enough for universal computation). We base this conjecture on two facts:

The only known universal elementary cellular automata figure at the top of this classification, and so do candidates such as rule 54 which figures right next to rule 110.

Universality seems to imply that a system should be capable of being

controlled by the inputs, which our classification suggests those at the bottom are not, as all of them look the same no matter what the input, and may not be capable of carrying information through the system toward the output.

The conjecture also seems to be in agreement with Wolfram's claim that rule 30 (as a class 3 elementary cellular automaton) may be, according to his Principle of Computational Equivalence (PCE), computationally universal. But it may turn out that it is too difficult (maybe impossible) to control in order to perform a computation because it behaves too randomly.

It is worth mentioning that this method is capable of capturing many of the subtle different behaviors a cellular automaton is capable of, which are heuristically captured by Wolfram's classes. The technique does not, however, disprove Wolfram's principle of irreducibility [2] because it is an a posteriori method. In other words, it is only by running the system that the method is capable of revealing the dynamical properties. This is no different from a manual inspection in that regard. However, it is of value that the method presented does identify a large range of qualitative properties without user intervention that other techniques (a priori techniques), including several analytical ones, generally seem to neglect.

4.7 Conclusion

We were able to clearly distinguish the different classes of behavior studied by Wolfram. By calculating the compressed lengths of the output of cellular automata using a general compression algorithm we found two clearly distinguishable main clusters and, upon closer inspection, two others with clear gaps in between. That we found two main large clusters seems to support Wolfram's Principle of Computational Equivalence (PCE) [2], which suggests that there is no essential distinction between the classes of systems showing trivial and nested behavior and those showing random and complex behavior. We have also provided a compression-based framework for phase transition detection, and a method to calculate an exponent capable of identifying and measuring the significance of other dynamical properties, such as sensitivity to initial conditions, presence of structures, and homogeneity in space or regularity in time. We have also formulated a conjecture with regard to a possible connection between its transition coefficient and the ability of a system to reach computational universality. As can be seen from the experiments presented in this paper, the compression-based approach and the tools that have been proposed are highly effective for classifying, clustering, and detecting several dynamical properties of abstract systems. Moreover, the

method does not depend on the system and can be applied to any abstract computing device or to data coming from any source whatsoever. It can also be used to calculate prior distributions and make predictions regarding the future evolution of a system.

Bibliography

[1] J. von Neumann, *Theory of Self-Reproducing Automata*, Urbana, IL: University of Illinois Press, 1966.

[2] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002.

[3] G. Braga, G. Cattaneo, P. Flocchini, and G. Mauri, "Complex Chaotic Behavior of a Class of Subshift Cellular Automata," *Complex Systems*, 7(4), 1993 pp. 269–296.

[4] G. Cattaneo and L. Margara, "Generalized Sub-Shifts in Elementary Cellular Automata: The 'Strange Case' of Chaotic Rule 180," *Theoretical Computer Science*, 201(1–2), 1998 pp. 171–187.

[5] F. Ohi and Y. Takamatsu, "Time-Space Pattern and Periodic Property of Elementary Cellular Automata—Sierpinski Gasket and Partially Sierpinski Gasket," *Japan Journal of Industrial and Applied Mathematics*, 18(1), 2001 pp. 59–73.

[6] M. Hurley, "Attractors in Cellular Automata," *Ergodic Theory and Dynamical Systems*, 10(1), 1990 pp. 131–140.

[7] F. Blanchard, P. Kurka, and A. Maass, "Topological and Measure-Theoretic Properties of One-Dimensional Cellular Automata," *Physica D: Nonlinear Phenomena*, 103(1–4), 1997 pp. 86–99.

[8] J. T. Baldwin and S. Shelah, "On the Classifiability of Cellular Automata," *Theoretical Computer Science*, 230(1–2), 2000 pp. 117–129.

[9] G. Braga, G. Cattaneo, P. Flocchini, and C. Quaranta Vogliotti, "Pattern Growth in Elementary Cellular Automata," *Theoretical Computer Science*, 145(1–2), 1995 pp. 1–26.

[10] K. Culik and S. Yu, "Undecidability of CA Classification Schemes," *Complex Systems*, 2(2) 1988 pp. 177–190.

[11] K. Sutner, "Cellular Automata and Intermediate Degrees," *Theoretical Computer Science*, 296(2), 2003 pp. 365–375.

[12] J.-C. Dubacq, B. Durand, and E. Formenti, "Kolmogorov Complexity and Cellular Automata Classification," *Theoretical Computer Science*, 259(1–2), 2001 pp. 271–285.

[13] F. Ohi, "Chaotic Properties of the Elementary Cellular Automaton Rule 40 in Wolfram's Class I," *Complex Systems*, 17(3), 2007 pp. 295–308.

[14] R. Cilibrasi and P. Vitanyi, "Clustering by Compression," *IEEE Transactions on Information Theory*, 51(4), 2005 pp. 1523–1545.

[15] G. J. Chaitin, *Algorithmic Information Theory*, 4th printing, Cambridge: Cambridge University Press, 1992.

[16] "Compress" from Wolfram *Mathematica* Documentation Center—A Wolfram Web Resource. http://reference.wolfram.com/mathematica/ref/Compress.html.

[17] J.-L. Gailly and M. Adler. "GZIP Documentation and Sources." Available as gzip-*.tar. ftp://prep.ai.mit.edu/pub/gnu/gzip.

[18] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the Institute of Radio Engineers*, 40(9), 1952 pp. 1098–1101.

[19] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, 23(3), 1977pp. 337–343.

[20] L. P. Deutsch. "DEFLATE Compressed Data Format Specification Version 1.3."(May 1996) http://www.rfc-editor.org/rfc/rfc1951.txt.

[21] H. Zenil. "Cellular Automaton Compressibility" from Wolfram Demonstrations Project—A Wolfram Web Resource. http://demonstrations.wolfram.com/CellularAutomatonCompressibility.

[22] J. P. Delahaye, "Voyageurs et baguenaudiers," *Pour La Science*, 238, 1997 pp. 100–104.

[23] C. S. Calude and M. A. Stay, "Most Programs Stop Quickly or Never Halt," *Advances in Applied Mathematics*, 40(3), 2008 pp. 295–308.

Chapitre 5

Image Characterization and Classification by Physical Complexity

From H. Zenil, J-P Delahaye and C. Gaucherel, *Image Characterization and Classification by Physical Complexity.*

5.1 Introduction

A method based on the theory of algorithmic information is presented, particularly based on the concept of Bennett's logical depth[1], to assess and quantify the information content of an image, providing a means for evaluating and classifying images by their organized complexity. Images have a number of properties containing information in the form of pixels. As a representation of an object, an image constitutes a description of the object capturing some features.

Algorithmic information theory[15, 4] formalizes the concepts of simplicity and randomness by means of information. Many applications of the theory of algorithmic information have been developed to date, for example [6, 12, 18, 19, 26, 27]. For a detailed survey see [17, 7]. None seems, however,

to have exploited the concept of logical depth, which may provide another useful complexity measure. Logical depth was originally identified with what is usually believed to be the right measure for evaluating the complexity of real-world objects such as living beings. Hence its alternative designation: physical complexity (by Bennett himself). This is because the concept of logical depth takes into account the plausible history of an object as an unfolding phenomenon. It combines the concept of the shortest possible description of an object with the time that it takes from the description to evolve to its current state. Unlike the application of the concept of algorithmic complexity by itself, the addition of logical depth results in a reasonable characterization of the organizational (physical) complexity of an object, as will be elaborated in the following sections.

The main hypothesis of this paper is that images can be used to determine the physical complexity of an object (or a part of it)[11] at a specific scale and level, or if preferred, to determine the complexity of the image containing information about an object. And as such, they cannot all be presumed to have the same history, but rather to span a wide and measurable spectrum of evolutions leading to patterns with different complexities, ranging from random-looking to highly organized. To test the applicability of the concept of logical depth to a real-world problem, we first approximate the shortest description of an image by way of current available lossless compression algorithms, then the decompression times are estimated as an approximation to the logical depth of the image. This allow us to assess a relative measure and to produce a classification based in this measure.

The paper is organized as follows: In section 5.2 the theoretical background that will constitute the formal basis for the proposed method is introduced. In Section 5.3 we describe the method and the battery of tests to evaluate it. Finally, in section 5.4 we present the results followed by the conclusions in 8.9.

5.2 Theoretical basis

5.2.1 Algorithmic complexity

The idea that information can be measured and described as a quantity using bits as units was first introduced by Shannon[21]. In computer science, objects can always be viewed as binary strings [1]. Thus we will refer to objects

1. Strings of characters, just like computer programs can also always be translated, with no loss of information, into bit strings.

and strings interchangeably in this discussion.

The complexity of a string of bits can be defined in terms of algorithmic complexity [2]. This is, given a program producing a string s, a machine can run the program and make a copy of s (in our case the image of the represented object). The relationship between Shannon's information theory and the theory of algorithmic complexity is described in [13]. The information content of a bit string can be defined as the length (in bits) of the smallest program which produces the string s.

This enables us to define a measure of randomness if the shortest algorithm able to generate s is not significantly shorter than s itself. In other words, there is no more economical way of communicating the information that it contains than by transmitting the string s in its entirety.

In algorithmic information theory a string is algorithmically random if it is incompressible. The difference in length between a string and the shortest algorithm able to generate it is the string's degree of complexity. A string of low complexity is highly compressible, as the information that it contains can be encoded in an algorithm much shorter than the string itself, while a string of high complexity is hard to compress because, in a fixed language, its shortest possible description is itself.

A classic example is a string composed by an alternation of bits such as $(01)^n$ that can be described by "n repetitions of 01". This string example can grow fast in length while the given description will only grow by about $\log_2(n)$. On the contrary, a random-looking string such as 011001011010110101 may not have a much shorter description than itself.

Algorithmic complexity is inversely related to the degree of regularity of a string. Any patterns in a string constitute redundancy: they enable one portion of the string to be recovered from another, allowing a more concise description. This is what is exploited by many lossless image compression algorithms, designed to find regularities in their bi-dimensional array.

The algorithmic complexity[15, 4, 16, 22] $K_U(s)$ of a binary string s with respect to a universal Turing machine U is defined as the binary length of the shortest program p of length $|p|$ that produces s as output:

$$K_U(s) = \min\{|p|, U(p) = s\}$$

Due to the halting problem (the problem of deciding whether, given a program and an input, the program will eventually halt when run with that

2. Also known under the names program-size complexity and Kolmogorov-Chaitin complexity.

input), a given program which computes only correct answers can compute the exact algorithmic complexity of at most a finite set of strings. For further details the reader can consult the classical textbooks in the field [3, 17]

Since $K(s)$ is the length of the shortest compressed form of s, i.e. the best possible compression (up to an additive constant), one can approximate K by compression means, using current state lossless compression programs. The better these programs compress, the better the approximations. The length of the compressed string in bits together with the decompressor in bits is an approximation of the shortest program generating the string. The length of the binary compressed version of s is an upper bound of its algorithmic complexity and therefore an approximation to $K(s)$.

5.2.2 Bennett's logical depth

A measure of the complexity of a string can be arrived at by combining the notions of algorithmic information content and time complexity. According to the concept of logical depth[1, 2], the complexity of a string is best defined by the time that an unfolding process takes to reproduce the string from its shortest description. The longer it takes the more complex. Hence complex objects are those which can be seen as "containing internal evidence of a nontrivial causal history."

Unlike algorithmic complexity, which assigns a high complexity to both random and highly organized objects placing them at the same level, logical depth assigns a low complexity to both random and trivial objects, thus being more in keeping with our intuitive sense of the complexity of physical objects because trivial and random objects are intuitively easy to produce, have no long history and unfold quickly. A clear, detailed explanation pointing out the convenience of the concept of logical depth as a measure of organized complexity as opposed to the usual plain algorithmic complexity is provided in [9].

A typical example that illustrates the concept of logical depth and its characterization as a measure of physical complexity is exemplified in sequence of fair coin tosses. Such a sequence would have high information content (algorithmic complexity) because the outcomes are random, but little value (logical depth) because they are easily generated and carry no message, no meaning. The string $1111\ldots1111$ is also logically shallow. Its minimal program, whilst very small, requires little time to evaluate. In contrast, the binary representation of the number π is not shallow, because although it is highly compressible (by any known formula producing π), the generating

algorithms require computational time to produce a number of digits of its expansion. A better example is Chaitin's Ω number[4], which digits encode the halting probability of a universal Turing machine, and which is known to be very deep since no computable process can expand Ω but for a finite number of digits[3, 10, 20].

Unlike in algorithmic complexity, in real-world computation physical resources usually do matter. A computation is seen as inherently difficult if computing it requires a large amount of time. In the real world, some objects such as a gas filling a room or a perfect crystal are intuitively trivial because they unfold in almost zero computing time, while others such as living beings contain internal evidence of a nontrivial causal history.

Bennett provides a careful development[1] of the notion of logical depth taking into account near-shortest programs as well as the shortest one, hence the significance value, for a reasonably robust and machine-independent measure. For finite strings, one of Bennett's formal approaches to the logical depth of a string is defined as follows:

Let s be a string and d a significance parameter. A string's depth at significance d, is given by

$$D_d(s)=\min\{T(p) : (|p| - |p'| < d) \wedge (U(p) = s)\}$$

the number of steps $T(p)$ in the computation $U(p) = s$, with $|p'|$ the length of the shortest program for s, (therefore $K(s)$). In other words, $D_d(s)$ is the least time T required to compute s from a d-incompressible program p on a Turing machine U.

Bennett's each of his three chained definitions of logical depth provided in [1] is closer to a definition in which near-shortest programs are taken into consideration. This is because a few bits more may give much shorter decompression time. The question of which precise definition to use is an aspect of further investigation. The simplicity of Bennett's first definition (the minimum computation time of some shortest program) is suitable to start our investigation as a practical approximation to this measure, by means of decompression times of compression algorithms. With the use of real-world compression algorithms, the decompression time of a compressed string is a lower bound of logical depth because the compressed version is unlikely to be the shortest (and, if so, there is no way to tell), the result may therefore presumably be larger than the minimum program and therefore so the decompression time. Hence the decompression time of the compressed version of a string is a lower bound of Bennett's of logical depth (from now on Bennett's

logical depth will be taken as it was given by Bennett in his definition 1 in paper [1]). Because the behavior of approximating the algorithmic complexity of a string with real-world compressors is asymptotic, that is, the better the compression algorithm the closest to the algorithmic complexity value of the string, using better compressors the decompression times also behave asymptotically semi-computing the logical depth from below. Bennett's definition also requires to pick the shortest among all programs producing a string, since this is obviously not feasible one can either try with several decompressors or try with the decompressor with better compression benchmarks and forcing the best algorithm to do its best (in a finite and reasonable time). From now on what we identified as the measure approximating the logical depth of a string s will be denoted by $D(s)$ with no need of a significance parameter due to the fact that we use Bennett's first simpler definition.

The concept of logical depth is an attempt to connect the description of an object with the time that it might take to produce it. Bennett's claim is that it is this time connecting the current state of an object with its plausible origin that is the appropriate measure of its complexity in physical terms. Bennett's main motivation was actually to provide a reasonable means of measuring the physical complexity of real-world objects, since the notion of logical depth stratifies them, placing those one would expect to be complex above those that one would expect to be simpler (although random strings are hardly compressible they can be quickly reproduced by a "print program" containing only a verbatim description of the data). Logical depth does this by taking into consideration the time that a process takes to produce the current state of an object from its plausible origin.

Algorithmic complexity and logical depth are intimately related. The latter depends on the former because one has to first find the shortest programs producing the string and then look for the one with shortest times, looking for the shortest programs is equivalent to approximating the algorithmic complexity of a string. While the compressed versions of a string are approximations of the algorithmic complexity of the string, decompression times are approximations of the logical depth of the string. These approximations depend on the compression algorithm. For algorithmic complexity the choice of universal Turing machine is bounded by an additive constant (invariance theorem) while logical depth by a linear polynomial[1].

5.3 Methodology

An image can be coded as a string s over a finite alphabet, say the binary alphabet–a black and white image. We will denote by K_c and D_c the approximations obtained by means of a compression algorithm c. When the algorithmic complexity $K(s)$ is approximated by a lossless compression algorithm, this approximation corresponds to an upper-bound of $K(s)$[17].

There seems to have been no previous attempt to implement an application of ideas based in Bennett's logical depth to a real-world problem. In order to assess the feasibility of an application of the concept to the problem of image characterization and classification by complexity we performed a series of experiments of gradually increasing sophistication, starting from fully controlled experiments and proceeding to the use of the best known compression algorithms over a larger dataset.

The battery of tests consisted of a series of images devised to verify different aspects of the methodology and a more realistic dataset, indicating whether the results were stable enough to yield the same values after each experiment repetition and whether they were consistent with the theory and consonant with the intuition of a complex vs. a simple object.

The first experiments consisted in controlling all the environmental parameters involved, from the data to the compression algorithm, in order to test the first attempts to calculate decompression times. A test to measure the correlation between image sizes (random vs. uniformly colored images) and decompression times was carried out. The results are in section 5.4.1.

A second test in section 5.4.1 consisted of a series of images meant to evaluate the change and magnitude of the decompression times when manipulating the internal structure of an image. That is, it served to verify that the decompression times decreased as expected when the content of a random image was artificially manipulated to make it more simple. This consisted of a set of images in which uniform structures consisting of large single-bit strings of a fixed size were randomly inserted into the images containing originally only pseudo-random generated pixels.

The framework consisted of using a toy compression program involving an algorithm grouping runs of the same bit replaced by a couple of values: the replaced bit followed by the number of times the bit was found. No further allowances were made, either for dealing with special cases or for detecting any other kinds of patterns. We wanted the algorithm and the data to be as simple as possible to be fully controlled in every detail to better understand the role of a structure of increasing size in the decompression

time. It consisted of the series of computer-generated random images shown in Figure 1.

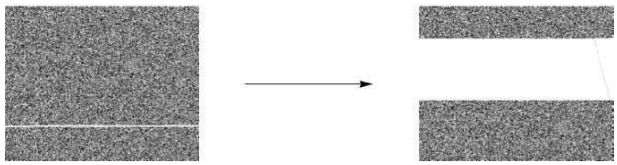

Figure 5.1: Inserting increasingly larger (white) regular blocks into an image containing originally randomly distributed black and white pixels.

This was a way of injecting structure into the images in order to study the behavior of the compression algorithm and assure control of the variables involved in order to better understand the next set of results.

A procedure described in 5.4.1 was devised to test the complementary case of the previous test to verify that the decompression times increased when the content of an image was artificially manipulated, transforming it from a simple state (an all-white image) to a more complex state by generating images with an increasing number of structures. A collection of one hundred images with an increasing number of straight lines randomly depicted was artificially generated (see Figure 2). The process led to interesting results described in 5.4.1 showing the robustness of the method.

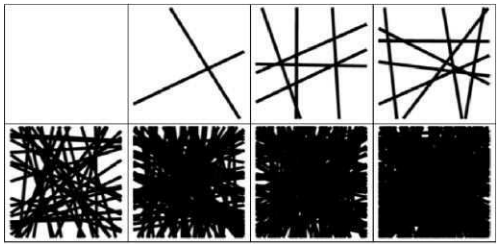

Figure 5.2: The number of lines grew as $2n^2$ with n the image number from 0 to 99. Depicting lines at this rate allowed us to start with a uniformly colored white image and end up with the desired, nearly all-black uniformly colored image in 100 steps.

Three more tests to calibrate the measure based on logical depth are proposed in 5.4.2 with three different series of images. The first two series of images were computer generated. The third one was a set of pictures of a real-world object. The description of the 3 series follows:

1. A series of images consisting of random points converted to black and white with a probability of a point's existing at any given location

having a certain value depending on a threshold. This can be seen as variation in information content since the ratio of black and white dots varies from high to low from image to image depending on the threshold value.

2. Cellular automaton with elementary rule number 30 (in Wolfram's enumeration[25]), then another image with the same automaton superposed upon itself (rotated 90°) followed by the same automaton to which was applied a function introducing random bits to 50 percent of the image pixels. We expect a greater standard deviation for the series of cellular automata because they come in pairs (meaning each image comes with its inverse), and permutations should be common and ought to be between these pairs because, intuitively, they should have the same complexity. This guess will be statistically confirmed in the results5.4 section.
3. A wall and the same wall but viewed from a closer vantage point.

The last test in 5.4.3 is the main result of the paper, showing the stability of method and the classification result. The experiment was performed on a dataset of randomly chosen pictures, 56 black-and-white pictures coming from different sources and representing objects of all kinds [3]. Some were pictures of actual objects produced by nature and humans, such as car plans, pictures of faces, handwriting, drawings, walls, insects, and so on. Others were computer-generated, such as straight lines, Peano curves, fractals, cellular automata, monochrome and pseudo-random generated images.

The selection of images in section 5.4.3 was made by hand bearing in mind the objective of spanning a large variety of objects covering a range of seemingly different complexities. We chose images of faces, people, engines and electronic boards, images singled out for their high degree of complexity, being each usually the result of a relatively long history, whether they were human artifacts or long-lived natural entities. The image bearing the name "table" is a table of numerical values, a computer spreadsheet, likely characterized by a significant level of complexity. Images tagged "people" are pictures of a group of people taken from a distance. The tag "inv" following picture names indicates that they are the color inversions of other images in the same dataset that we expected would be close in complexity, and they are presented side-by-side with their non-inverted versions. "Writing" refers to handwriting by a human being, which should also have a significant level of complexity, a level of complexity approaching that of other handwritten

3. The images are available online under the paper title at http://www.algorithmicnature.org

pieces such as the image tagged "formulae" which depicts formulae rather than words. "Watch" is the internal engine of a watch. "Cpu A" is a picture of a microprocessor showing less detail than "cpu B". "Escher" is a painting of tiles by Escher. "Paper" is a corrugated sheet of paper. "Shadows B" are the shadows produced by sea tides. "Symmetric B" is "shadows B" repeated 4 times. "Symmetric A" is "symmetric B" repeated 4 times. "Rectangle C" is "rectangle B" repeated 4 times. Those images tagged "Peano curve" are the space filling curves. "Periodic" and "Alternated" are of a similar type, and we thought they would have low complexity, being simple images. "Random" was a computer-generated image, technically pseudo-random. This random-generated picture will illustrate the main known difference between algorithmic complexity and logical depth. Namely, based on its algorithmic (program-size) complexity the random image should be ranked at the top, whereas its logical depth would place it at the bottom. All pictures were randomly chosen from the web, transformed to B&W and reduced to 600 × 600 pixels [4].

5.3.1 Towards the choice of lossless compression algorithm

Deflate [5] is a compliant lossless compressor and decompressor available within the zlib package. The Deflate compression compresses data using a combination of the Lempel-Ziv coding algorithm[28] and the Huffman coding[14]. It is one of the most widely-used compression encoding systems.

The Huffman coding assigns short codewords to those input blocks with high probabilities and long codewords to those with low probabilities. In other words, the compressor encodes more frequent sequences with a few bytes and spends more bytes only on rare sequences.

The Lempel-Ziv coding algorithm builds a dictionary and encodes the string by blocks using symbols in the dictionary. The Lempel-Ziv algorithm leads to actual compression when the input data is sufficiently large and there is sufficient redundancy (patterns) in the data[24]. Lempel-Ziv compression algorithm provides upper bounds to K.

4. Unlike the printed version, if seen in electronic form the images can be zoomed in, allowing better visualization. They are also available online under the paper title at http://www.algorithmicnature.org

5. RFC1951: Deflate Compressed Data Format Specification version 1.3 http://www.w3.org/Graphics/PNG/RFC-1951.

5.3.2 Compression method

Since digital images are just strings of values arranged in a special way, one can use image compression techniques to approximate the algorithmic complexity of an image. And through its image, the algorithmic complexity of the object represented by the image (at the scale and level of detail of the said image).

A representative sample of lossless image-compression algorithms (GIF, TIFF and JPG 2000) was tested in order to compare the decompression runtimes of the testing images. Although we experimented with these lossless compression algorithms, we ended up choosing PNG for several reasons, including its stability and the flexibility afforded by the ability to use open-source developed optimizers for further compression.

The Portable Network Graphics (PNG) is a bitmapped image format that employs lossless data compression. It uses a 2-stage compression process, a pre-compression or filtering, and Deflate. The filtering process is particularly important in relating separate rows, since Deflate alone has no understanding that an image is a bi-dimensional array, and instead just sees the image data as a stream of bytes.

There are several types of filters embedded in image compression algorithms which exploit regularities found in a stream of data based on the corresponding byte of the pixel to the left, above, above and to the left, or any combination thereof, and encode the difference between the predicted value and the actual value.

$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$
$\mathbf{x}_4$	$\mathbf{x}_9$	$\mathbf{x}_5$
$\mathbf{x}_6$	$\mathbf{x}_7$	$\mathbf{x}_8$

Figure 5.3: Image pixel neighborhood. By applying different filters a lossless image compression algorithm uses the data contained in the pixels $x_1, \ldots, x_8$ to predict the value of another pixel x_9 that may save space, allowing the compression of the said image without losing any information. The number of combination tested is finite and limited by the compression algorithm. Yet one can optimize the search of a successful combination by setting the compression algorithm to try harder and spend more time trying to better compress the data.

Compression can be further improved by so-called PNG-optimizers using more filter methods and several other lossless data compression algorithms.

Among these optimizers are Pngcrush [6] and AdvanceCOMP [7], two of the most popular open-source optimizers. They tried various compression methods and were able to reduce the PNG files by about 10 to 20% of their original length. AdvanceCOMP recompresses PNG files (and other file formats) using the Deflate 7-Zip implementation. The 7-Zip Deflate encoder effectively extends the scope of Deflate further by performing a much more detailed search of compression possibilities at the expense of significant further processor time spent searching, which for this experiment was not a matter of concern. 7-Zip Deflate also uses the LZMA algorithm, an improved and optimized version of the LZ77[28] compression algorithm. The LZMA [8] algorithm divides the data into packets, each packet describing either a single byte or an LZ77 sequence with its length and distance implicitly or explicitly encoded [9].

As a sort of verification, we ran a popular zip-based commercial compressor, set to the maximum possible compression, over the already compressed and optimized files. The zip-based archiver was unable to further compress any of the files (they were actually always a little larger in size).

5.3.3 Using decompression times to estimate complexity

Inflate is the decoding process that takes a Deflate bit stream for decompression and correctly produces the original full-size data file. To decode an LZW-compressed file, one needs to know the dictionary encoding the matched strings in the original data to be reconstructed. The decoding algorithm for LZ77 works by reading a value from the encoded input and outputting the corresponding string from the shorter file description.

It is this decoding information that Inflate takes when importing a PNG image for display, so the lengthier the directions for decoding the longer the time it takes. We are interested in these compression/decompression processes, particularly the compression size and the decompression time, as an approximation of the algorithmic complexity and the logical depth of an image.

6. Syntax example: pngcrush -reduce -brute -e ".compressed.png" /testimages/*.png. More info: http://pmt.sourceforge.net/pngcrush/

7. Syntax example: e.g. advdef -z -4 *.png ('4' indicating the so-called "insane" compression according to the developers). More info: http://advancemame.sourceforge.net/

8. More technical details are given inhttp://www.7-zip.org/7z.html.

9. A useful website showing a benchmark of compression algorithms is at url-http://tukaani.org/lzma/benchmarks.html.

The decompression directions for trivial or random-looking objects are simple to follow, with the decompression process taking only a small amount of time, simply because the compression algorithm either compresses very well and the decompression is just straightforward (trivial case) or does not compress at all (random case). Longer runtimes, however, are usually the result of a process following a set of time-consuming decompression instructions, hence a complex process.

5.3.4 Timing method

The execution time was given by the *Mathematica* function Timing [10]. The function Timing evaluates an expression and returns a list of the time used in seconds, together with the result obtained. The function includes only CPU time spent in the *Mathematica* kernel.

The fact that several processes run concurrently in computing systems as part of their normal operation is one of the greatest difficulties faced in attempting to measure with accuracy the time that a decompression process takes, even when it is isolated from all other computer processes. This instability is due to the fact that the internal operation of the computer comes in several layers,mostly at the operating system level. In order to avoid measurement perturbations as much as possible, several stabilizing measures were undertaken:

- Most computer batch processes and operating system services were disabled, including services such as wireless and bluetooth and energy saving features, such as the hard drive sleeping and display dimming features [11].
- The microprocessor was warmed up before each experiment by running an equivalent process (e.g. a mock experiment run) before running the actual one in order to have the fan and everything else already working at a high rate (like preheating an oven).
- The cache memory was cleared after each function call using the *Mathematica* function ClearSystemCache. No history was saved in RAM memory.

10. Timed on two different computers for validation. On a MacBook Intel Core 2 Duo 2GHz, 2048MB DDR2 667Mhz with a solid-state drive (SSD) and on MacBook Pro Intel Core 2 Duo 2.26Ghz, 4096MB DDR3 1067Mhz with a traditional hard disk drive (HDD), both running Mac OS X Version 10.6.1 (Snow Leopard). The MacBook Pro was always twice as fast on average.

11. To understand the number of layers of complexity involved in a modern computing system see Tanenbaum's book on operating systems[23].

– A different order of images was used for each experiment run. The result was averaged with at least 30 runs, each compressing the images in a different random order. This helped to define a confidence level on the basis of the standard deviation of the runs, when they are normally distributed. A confidence level with which we were satisfied and at which we arrived in a reasonable amount of time. Further runs showed no further improvement. Measurements stabilization was reached after about 20 to 30 runs.

The presence of some perturbations in the time measure values were unavoidable due to lack of complete control over all the computing system parameters. As the following battery of tests will show, one can reduce and statistically curtail the impact of this uncertainty to reasonable levels.

5.4 Results

5.4.1 Controlled experiments

Image size uncertainty variation

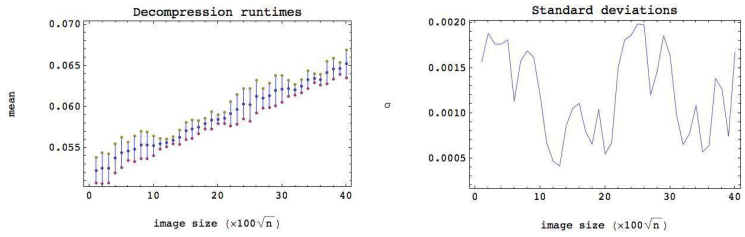

Figure 5.4: Decompression times and standard deviations for monochromatic images increasing in size.

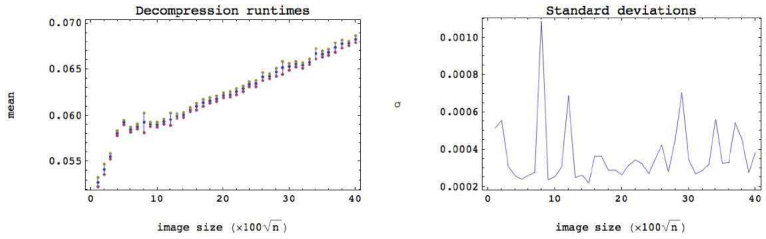

Figure 5.5: Decompression times and standard deviations for images with (pseudo) random noise increasing in size.

Figures 4 and 5 show that decompression times using the PNG algorithm and the optimizers increase in linear fashion when image sizes increase linearly. The distributions of standard deviations in the same figures show no particular tendency other than suggesting that the standard deviations are likely due to random perturbations and not to a bias in the methodology.

A comparison with Figure 6 shows that standard deviations for images containing random noise remained always low, while for uniformly colored images standard deviations were significantly larger.

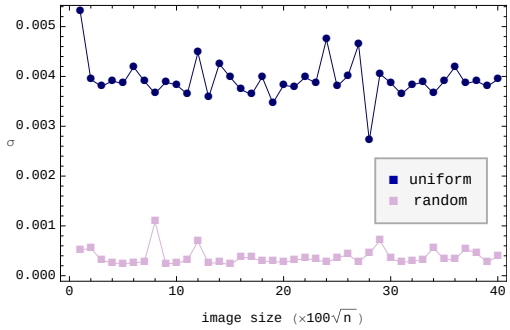

Figure 5.6: For uniformly colored images standard deviations of decompression times were larger in average. For random images the standard deviations were smaller and compact. The size of the image seemed to have no impact in the behavior of the standard deviations for either case.

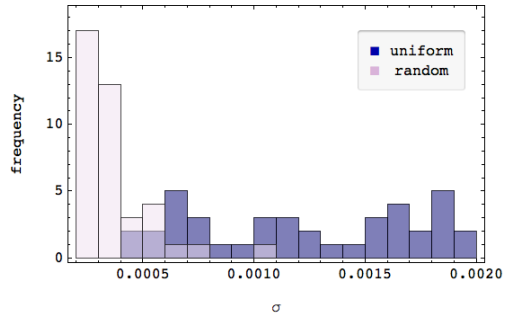

Figure 5.7: Histogram of the standard deviations for uniformly colored and random images increasing in size.

Fully controlled test

For this test only, we used a toy compression algorithm designed to be as simple as possible in order to estimate the uncertainty due to system

perturbations. We devised a test in which we would have full control of the data and the compression algorithm. By understanding the processes we would be able to predict compression rates and decompression uncertainties better, and quantify the uncertainty of the measurement of decompression times in the general case (i.e. using other compression algorithms).

It was found that the algorithmic complexity approximation (the file size) decreased linearly at a rate of 1990 bits per image, the same as the number of regular inserted bits per image, taking about $\log_2(2000)$ to encode the compressed regular string, as theory (Shannon information) may have predicted. One can also predict the decompression runtime by calculating the slope of the decompression rate. Each insertion of 2000 regular bits into the random image took 0.00356 seconds less each time, fitting the estimate computed by the rate ratio.

The standard deviations behavior suggest that the more random an image the less stable the decompression time and the more regular the more stable. This may be explained by the number of operations that the toy compression algorithm uses for encoding a regular string. In the extreme case of a uniformly colored image, the code comes up with a single loop operation to reproduce the image, taking only one unit of time. On the other hand, when an image is random, wholly or partially, the number of operations is larger because the algorithm finds small patterns everywhere, patterns that have a timing cost per operation performed. Each loop operation using the Table function in *Mathematica* takes 0.000025 seconds on average, while each iteration takes only about 0.00012 seconds on average, suggesting that it is the number of operations that determines the runtime.

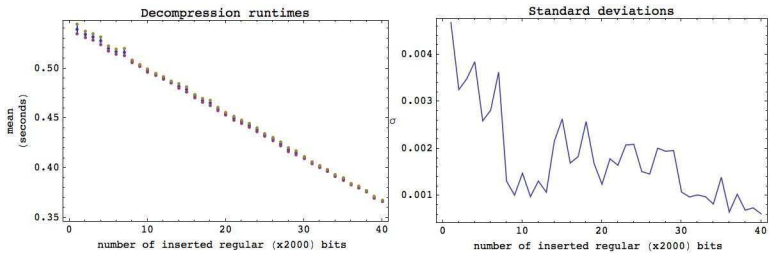

Figure 5.8: Using the toy compression algorithm, the larger the white region in an image with random noise the shorter the decompression runtime and the lower the standard deviations.

The standard deviation of the mean 0.0000728 is smaller than the average of the standard deviations of each point, which is 0.000381. This shows that runtime perturbations were not significant enough.

Using the PNG compression algorithm

This time we proceeded to apply the PNG compressor algorithm together with the optimizers (Pngcrush and AdvancedCOMP) to the same computer-generated images with random noise used in the previous section, confident that we understood the variables involved in the previous experiment, and that everything seemed to be controlled.

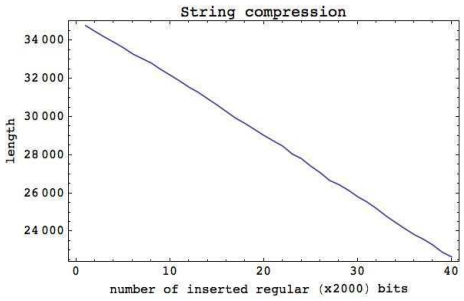

Figure 5.9: Uniformly colored images case compression rate. As for the controlled experiment with the toy compression algorithm using the PNG compression algorithm, the compressed length of the images also decreased in a predictable way as one would have expected for this controlled experiment. And they did so at a rate of 310 bits per image.

Figure 9 shows that the PNG [12] compression algorithm followed the same trend as the toy compression algorithm, namely the more uniformity in a random image the greater the compression. PNG however achieved greater compression ratios compared to the toy compression algorithm, for obvious reasons.

In Figure 9 one can see that there are some minor bumps and hollows all along. Their deviation does not seem however significant from a uniformly distribution, as shown in Figure 10, suggesting bias toward no particular direction. As was the case with the toy compression algorithm, by using the PNG algorithm algorithmic complexity decreased as expected–upon the insertion of uniform strings.

Increasing complexity test

The first thing worth noticing in Figure 11 (left) is that the standard deviations remained the same on average, suggesting that they were not

12. with Pngcrush and AdvanceCOMP

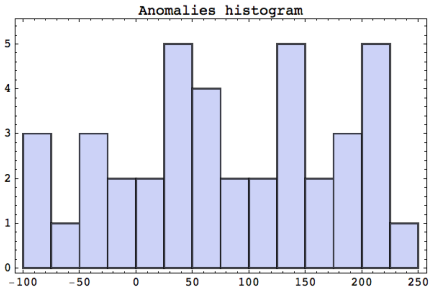

Figure 5.10: Anomalies found in the decompression times deviating from the main slope seem to behave randomly, suggesting no methodological bias.

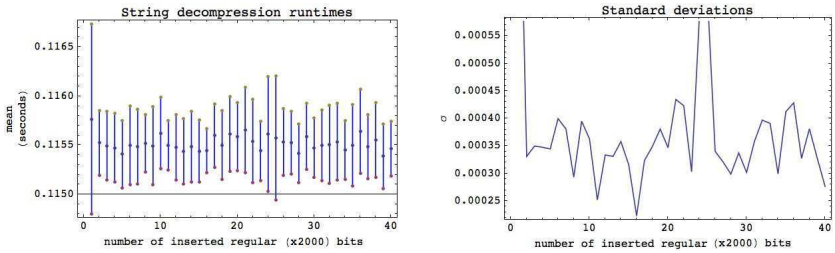

Figure 5.11: Random images produce smaller standard deviations (notice the plot scale). But unlike the toy case, decompression times remained statistically the same despite an increase in image size.

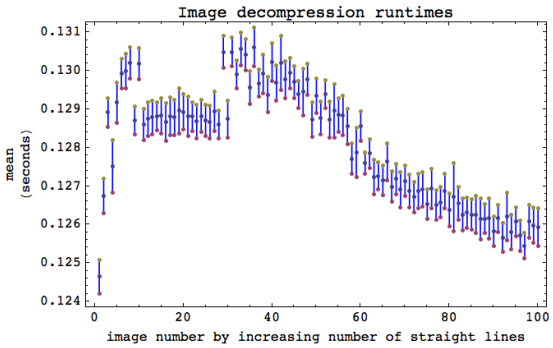

Figure 5.12: Compressed path: Going from all white to all black by randomly depicting black lines.

related to the image, but to external processes. The process of randomly depicting lines seems to have a low maximum limit of complexity, which is rapidly reached. Nevertheless, eight bins of images with significantly different decompression time values, i.e. with non-overlapping standard deviations, were identified. What Figure 12 suggests is that increasing the number of

lines can only lead to a limited maximum degree of complexity bounded by a convex curve. The eight significantly different images have been selected according to jumps that were maximizing their differences: $n = \Delta D/\sigma$ with $\Delta = \max\{D(I_i)|i \in \{1,\ldots,n\}\} - \min\{D(I_i)|i \in \{1,\ldots,n\}\}$, with I_i each image in the set, and D the logical depth.

For the discontinuity in the graph, one hypothesis is that the compression algorithm reaches a threshold favoring some regularities over others, producing jumps in the decompression times. For a certain quantity of randomly depicted lines, the limit remains stable for a while once reached, until the moment when the increasing number of lines depicted fill up the space and the configuration reaches its lowest complexity by decompression time, when it begins to approach a phase in which it resembles a uniformly colored image.

The decompression time depicted in Figure 12 turned out to be very interesting suggesting what one might expect for this kind of experiment: A path traced between two monochromatic (fully colored) images, an initial all-white image succeeded by images of increasing complexity comprising random lines, and ending at the horizontal departure line in a final monochromatic almost all-black image, when the space becomes entirely filled with black lines.

5.4.2 Calibration tests

The following are the classifications of series 1, 2 and 3 according to their decompression runtimes as described in the methodology section 5.3.

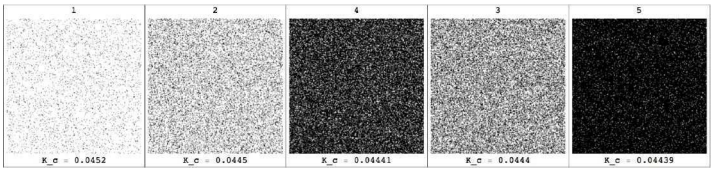

Figure 5.13: Series 1 classification: Random points with different densities.

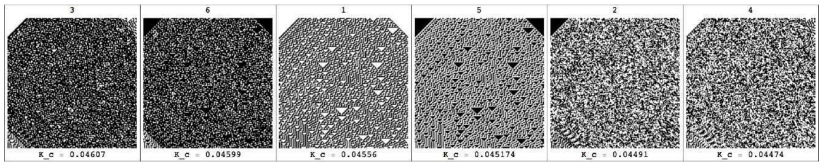

Figure 5.14: Series 2 classification: Cellular automata superpositions, rotations and inversions.

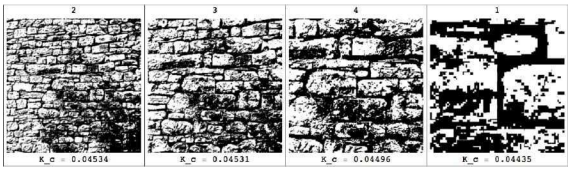

Figure 5.15: Series 3 classification: Zooming in a wall.

series number	*description*	*Std. deviation*
2	*cellular automata*	0.000556
3	*wall zooming*	0.00046
1	*random points*	0.000349

Table 5.1: Standard deviations per series sorted from greatest to lowest standard deviation.

- Series 1 classification (Figure 13): Images labeled 1 and 2 never came last, while images 4 and 5 never came first. The first half of the images remained the same run after run, while the second half also remained stable, with some occasional permutations within the intermediate elements of each half, but never with the extremes exchanging places.
- Series 2 classification (Figure 14): A cellular automaton superposed upon itself rotated 90°, with random bits inserted. The runs not only mostly never changed, the procedure consistently sorted the images into pairs, grouping images with similar characteristics (to which the same function was applied). The classification seems consonant with what one may see as the more random vs. less random. All of them had low physical complexity in general, given their values, which is in accordance with what is believed about the cellular automaton rule 30, i.e. that it is known to behave randomly[25], with some regular structures on the left.
- Series 3 classification (Figure 15): A wall and the same wall but viewed from a closer vantage point. It turned out to be highly stable after several runs, sorting the series of images from the farthest to the closest picture of the wall, suggesting that pictures taken from a greater distance captured more of the structure of the wall, while zooming in without also increasing the resolution meant losing detail and structure. The closest picture always came last, while the farthest always came first. There were only occasional permutations between those in between.

As was expected (see section 5.3), a greater standard deviation for the series of cellular automata was found due to the image pairing. Each image occurred in tandem with its inverse, but permutations were common–and expected–between members of pairs, which intuition tells us ought to have the same complexity. The expected permutations produced a larger standard deviation.

5.4.3 Compression length ranking

The following experiments were carried out as described in section 5.3 using 56 black-and-white images spanning a range of different kind of objects each seemingly having different complexity which we could intuitively gauge more or less accurately [13]. Each image has a very short description, followed by the image itself and by the approximated values of K_c (Figure 16) and D_c (Figure 17) for our general compression algorithm c.

The classification in Figure 16 presents the images ranked according to their compressed lengths using the PNG image compression together with the PNG optimizers. It goes from larger to smaller, and as can be seen, the more random-looking or highly structured, the better classified, while trivial images come last. One can verify that the procedure is invariant to simple transformations, such as inversions and complementations, since images and their symmetric versions are always next (or close) to each other, indicating that the compression algorithm behaves as one would expected (that is, that inverting colors for example, has no impact in the resultant compressed length). One would hardly say, however, that a random image is physically complex (random processes, like a gas filling a room, unfold in almost zero time and seem to require no great computational power), which is why plain algorithmic complexity does not help to distinguish between complexity associated with randomness and complexity associated with highly structured objects.

5.4.4 Decompression times ranking

The classification in Figure 17 goes from greater to smaller logical depth based on the decompression runtimes. The number at the bottom is the time in seconds that the PNG algorithm took to decompress the images when applied to their compressed versions. Images we gauged to have the

13. The images are available online under the paper title at http://www.algorithmicnature.org

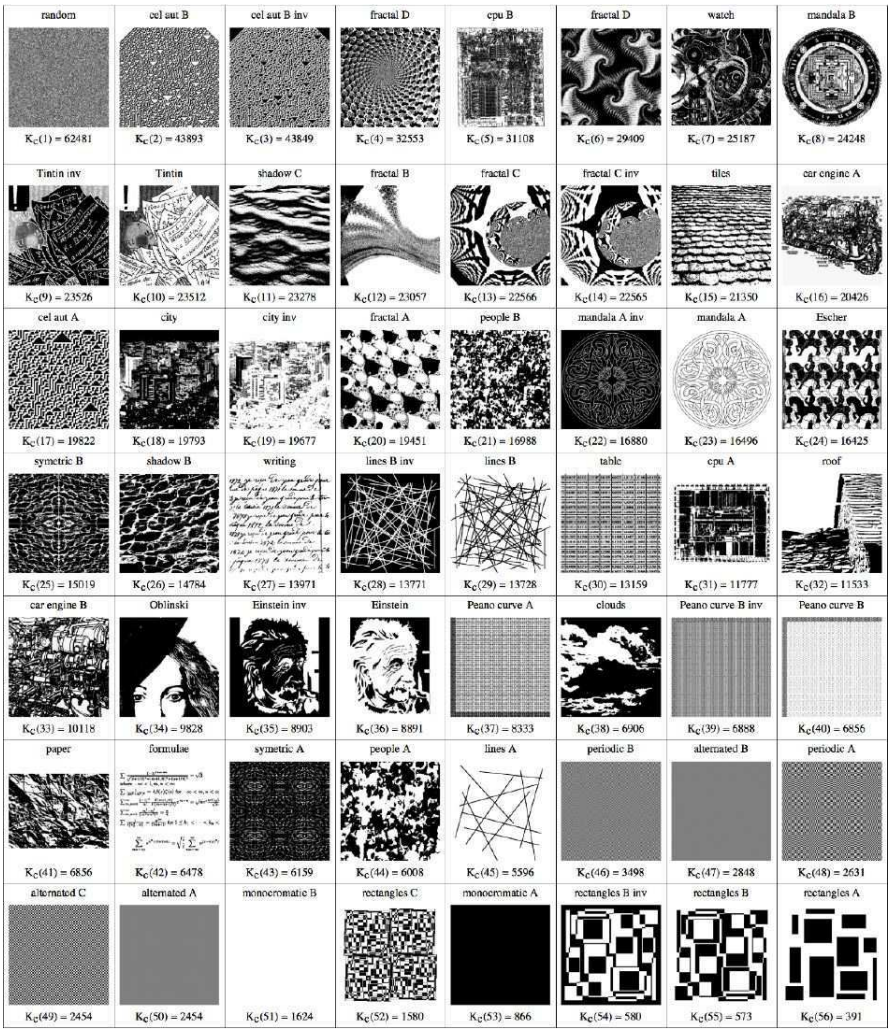

Figure 5.16: Ranking by compressed lengths: $K_c(i_n) \leq K_c(i_{n+1})$ with $K_c(i)$ the approximation to the algorithmic complexity $K(i)$ of the image i by means of the PNG compression algorithm aided by Pngcrush and AdvanceCOMP.

highest physical complexity come first. It is also worth mentioning that the images and their inverted versions remained close to each other, meaning that they were always in the same complexity group, which is also just what we expected. This also indicates the soundness of the procedure, since images and their inversions should be equal in complexity, and therefore equal in complexity to the object as well, at a commensurate scale and level of detail.

Figure 18 shows some of the jumps seen before in the experiments in section 5.4.1. We think they may be due to the behavior of the compression

Figure 5.17: Ranking by decompression times: $D_c(i_n) \leq D_c(i_{n+1})$ with $D_c(i)$ the approximation to the logical depth $D(i)$ of the image with number i according to the indexing from the previous classification for K_c, by means of the PNG compression algorithm aided by Pngcrush and AdvanceCOMP.

algorithm. The compression algorithm applies several filters and it may favor some regularities over others that are better (faster) decoded after certain threshold producing these jumps.

Images were grouped in 8 significantly different groups (with different decompression times and therefore seemingly different logical depth). Formally, $\bar{x}(g_i) \pm \sigma(g_i) > \bar{x}(g_j) \pm \sigma(g_j)$ for any two different group indexes $i, j \in \{A, B, \ldots, H\}$.

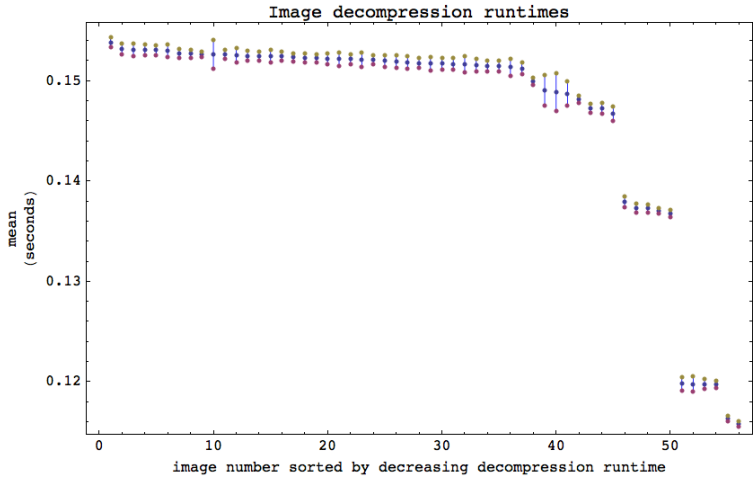

Figure 5.18: Mean and standard deviations of the decompression times of the 56 images.

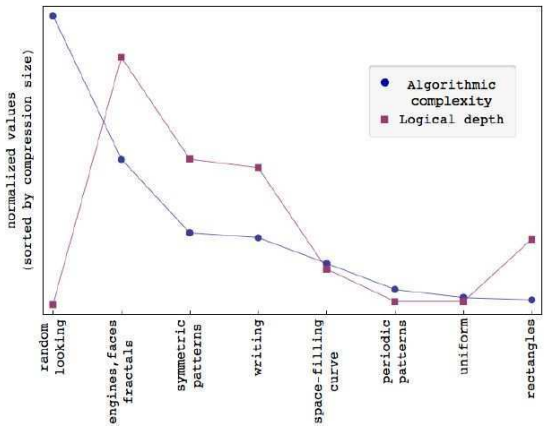

Figure 5.19: Rank comparison between K_c and D_c as functions of the chosen images having significantly different D_c values.

The average difference between the largest and the smallest K_c value was about 62090 bits, while the average difference between the largest and the smallest D_c was 0.095 seconds. The largest calculated compressed image size (image 1) was 159.8 times larger than the shortest compressed image size (image 56). The largest evaluated decompression time (image 5) was 2.46 times larger than the shortest calculated decompression time (image 53).

No significant statistical correlation between K_c and D_c was found, indicating that K_c and D_c are actually two different measures. Figure 19 and 20 illustrate the classification values from K_c to 8 significant different decom-

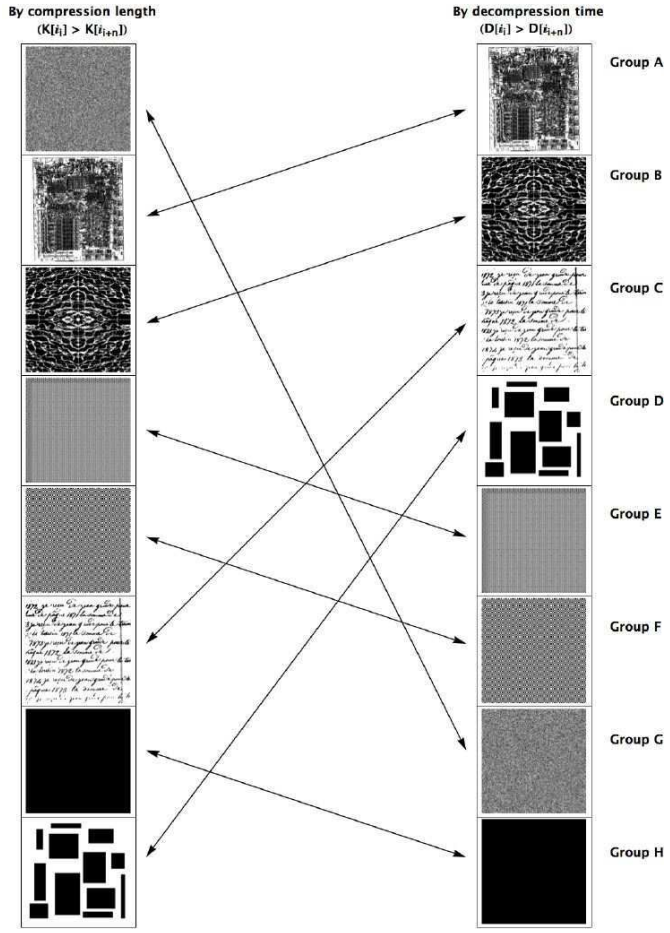

Figure 5.20: K to D mapping by groups indicating the complexity group (by logical depth) to which each image instance on the right belongs.

pression time (D_c) groups according to each of the two measures. The ranking of images based on their decompression times correspond to the intuitive ranking resulting from a visual inspection, with things like microprocessors, human faces, cities, engines and fractals figuring at the top as the most complex objects, as shown in Figure 21 (group A); and random-looking images, which ranked high by algorithmic complexity, were ranked low (group G) according to the logical depth expectation, classified next to trivial images such as the uniformly colored (group H), indicating the characteristic feature of the measure of logical depth. A gradation of different complexities can be found in the groups between, gradually increasing in complexity from bottom to top.

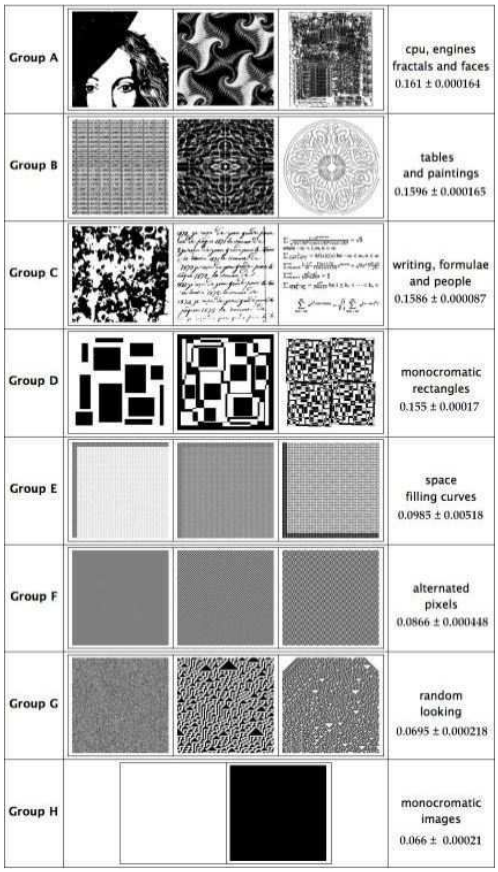

Figure 5.21: Significant different groups by decreasing decompression time.

5.5 Conclusions and further work

Extensive experiments were conducted. Along the way we think we have shown that:

1. Ideas in the spirit of Bennett's logical depth can be implemented to approach a real-world characterization and classification problem other than using the concept of algorithmic complexity alone, distinguishing only between random and trivial objects but not separating random from structured objects.
2. After studying several cases and tested several compression algorithms, the method described in this paper has shown to work and to be of use for identifying and classifying images by their apparent physical complexity.
3. The procedure described herein constitutes an unsupervised method for

evaluating the information content of an image by physical complexity.

4. The method of decompression times yields a reasonable measure of complexity that is different from the measure obtained by considering algorithmic complexity alone, while being in accordance with one's intuitive expectations of greater and lesser complexity.

Further to investigate is how to improve this procedure to accurately follow Bennett's stricter definitions of Logical Depth. For example, it could be that by allowing a couple of bits more in the compressed versions of the images one gets shorter decompression times, and we think experiments in this regards are possible. To stay in the spirit of Bennett one may use several compressors $c_1, \ldots, c_n$ on strings $s_1, \ldots, s_m$ and compute $D_{c_i(s_j)}$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$.

Then, an object s_h is assumed to be more complex than an object s_k if $D_{c_i(x_j)} > D_{c_i(x_k)}$ for all $1 \leq i \leq n$ and thus obtain a partial order from which a total order could be obtained by calculating the (e.g. harmonic) average of the decompression times.

Bibliography

[1] C.H. Bennett, Logical Depth and Physical Complexity in Rolf Herken (ed) The Universal Turing Machine–a Half-Century Survey Oxford University Press 227-257, 1988.

[2] C.H. Bennett, How to define complexity in physics and why. In Complexity, entropy and the physics of information, Zurek, W. H., Addison-Wesley, Eds. SFI studies in the sciences of complexity, p 137-148, 1990.

[3] C.S. Calude, Information and Randomness: An Algorithmic Perspective (Texts in Theoretical Computer Science. An EATCS Series) Springer, 2nd. edition, 2002.

[4] G.J. Chaitin A Theory of Program Size Formally Identical to Information Theory, J. Assoc. Comput. Mach. 22, 329-340, 1975.

[5] G.J. Chaitin, Algorithmic information theory, IBM Journal of Research and Development, v.21, No. 4, 350-359, 1977.

[6] X. Chen, B. Francia, M. Li, B. Mckinnon, A. Seker, Shared Information and Program Plagiarism Detection IEEE Trans. Information Theory, 2004.

[7] R. Cilibrasi and P. Vitányi, Clustering by compression IEEE Trans. on Information Theory, 51(4), 2005.

[8] J.P. Delahaye and H. Zenil, On the Kolmogorov-Chaitin complexity for short sequences, in Cristian Calude (eds) Complexity and Randomness: From Leibniz to Chaitin, World Scientific, 2007.

[9] J.P. Delahaye, Complexité aléatoire et complexité organisée Editions Quae, 2009.

[10] R.G. Downey and D. Hirschfeldt, Algorithmic Randomness and Complexity, Springer Verlag, 2010.

[11] C. Gaucherel, Influence of spatial structures on ecological applications of extremal principles Ecological Modelling, 193: p. 531-542, 2006.

[12] S. Goel and S.F. Bush, Kolmogorov Complexity Estimates for Detection of Viruses in Biologically Inspired Security Systems: A Comparison with Traditional Approaches Complexity, vol. 9, no. 2, 2003.

[13] P.D. Grunwald, P. Vitányi, Kolmogorov complexity and Information Theory Journal of Logic, Language and Information, vol. 12, issue 4, pp 497-529, 2003.

[14] D. A. Huffman, A Method for the Construction of Minimum Redundancy Codes Proceedings of the IRE, Vol. 40, pp. 1098-1101, 1952.

[15] A. N. Kolmogorov, Three approaches to the quantitative definition of information Problems of Information and Transmission, 1(1):1–7, 1965.

[16] L. Levin, Laws of information conservation (non-growth) and aspects of the foundation of probability theory, Problems of Information Transmission, 10(3):206–210, 1974.

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

[18] M. Li, J. Badger, X. Chen, S. Kwong, P. Kearney and H. Zhang, An Information-Based Sequence Distance and Its Application to Whole Mitochondrial Genome Phylogeny Bioinformatics 17(2):149154, 2001.

[19] M. Li, X. Chen, X. Li, B. Ma, P. Vitányi, The similarity metric, in Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 863-872, 2003.

[20] A. Nies, Computability and Randomness, Oxford University Press, 2009.

[21] L.E. Shannon, A mathematical theory of communication Bell Systems Technical Journal, vol. 27, 1948.

[22] R.J. Solomonoff. A formal theory of inductive inference: Parts 1 and 2 Information and Control, 7:1–22 and 224–254, 1964.

[23] A.S. Tanenbaum and A.S. Woodhull, Operating Systems Design and Implementation (3rd Edition), 2006.

[24] T.A. Welch, A Technique for High-Performance Data Compression Computer, vol. 17, no. 6, pp. 8-19, June 1984.

[25] S. Wolfram, A New Kind of Science Wolfram Media, 2002.

[26] H. Zenil, Compression-based investigation of the dynamical properties of cellular automata and other systems journal of Complex Systems, vol. 18:4, 2010.

[27] H. Zenil and J.P. Delahaye, "On the Algorithmic Nature of the World," in Gordana Dodig-Crnkovic and Mark Burgin (eds), Information and Computation, World Scientific, 2010.

[28] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, 23(3), pp. 337343, May 1977.

Chapitre 6

Sloane's Gap : Do Mathematical and Social Factors Explain the Distribution of Numbers in the OEIS ?

Forthcoming in N. Gauvrit, J-P. Delahaye and H. Zenil, *Le Fossé de Sloane*, Mathématiques et Sciences Humaines - Mathematics and Social Sciences.

6.1 Introduction

The Sloane encyclopedia of integer sequences[10] [1] (OEIS) is a remarkable database of sequences of integer numbers, carried out methodically and with determination over forty years[3]. As for May 27, 2011, the OEIS contained 189,701 integer sequences. Its compilation has involved hundreds of

1. The encyclopedia is available at: http://oeis.org/, last consulted 26 may, 2011.

mathematicians, which confers it an air of homogeneity and apparently some general mathematical objectivity–something we will discuss later on.

When plotting $N(n)$ (the number of occurrences of an integer in the OEIS) two main features are evident:
(a) Statistical regression shows that the points $N(n)$ cluster around $k/n^{1.33}$, where $k = 2.53 \times 10^8$.
(b) Visual inspection of the graph shows that actually there are two distinct sub-clusters (the upper one and the lower one) and there is a visible gap between them. We introduce and explain the phenomenon of "Sloane's gap."

The paper and rationale of our explanation proceeds as follows:
We explain that (a) can be understood using algorithmic information theory. If U is a universal Turing machine, and we denote $m(x)$ the probability that U produces a string x, then $m(x) = k2^{-K(x)+O(1)}$, for some constant k, where $K(x)$ is the length of the shortest description of x via U. $m(.)$ is usually refered to as the Levin's universal distribution or the Solomonoff-Levin measure [7]. For a number n, viewed as a binary string via its binary representation, $K(n) \leq \log_2 n + 2 \log_2 \log_2 n + O(1)$ and, for most n, $K(n) \geq \log_2 n$. Therefore for most n, $m(n)$ lies between $k/(n(\log_2 n)^2)$ and k/n. Thus, if we view OEIS in some sense as a universal Turing machine, algorithmic probability explains (a).

Fact (b), however, is not predicted by algorithmic complexity and is not produced when a database is populated with automatically generated sequences. This gap is unexpected and requires an explanation. We speculate that OEIS is biased towards social preferences of mathematicians and their strong interest in certain sequences of integers (even numbers, primes, and so on). We quantified such a bias and provided statistical facts about it.

6.2 Presentation of the database

The encyclopedia is represented as a catalogue of sequences of whole numbers and not as a list of numbers. However, the underlying vision of the work as well as its arrangement make it effectively a dictionary of numbers, with the capacity to determine the particular properties of a given integer as well as how many known properties a given integer possesses.

A common use of the Sloane encyclopedia is in determining the logic of a sequence of integers. If, for example, you submit to it the sequence 3, 4, 6, 8, 12, 14, 18, 20..., you will instantly find that it has to do with the sequence

of prime augmented numbers, as follows: 2+1, 3+1, 5+1, 7+1, 11+1, 13+1, 17+1, 19+1...

Even more interesting, perhaps, is the program's capacity to query the database about an isolated number. Let us take as an example the Hardy-Ramanujan number, 1729 (the smallest integer being the sum of two cubes of two different shapes). The program indicates that it knows of more than 350 sequences to which 1729 belongs. Each one identifies a property of 1729 that it is possible to examine. The responses are classified in order of importance, an order based on the citations of sequences in mathematical commentaries and the encyclopedia's own cross-references. Its foremost property is that it is the third Carmichael number (number n not prime for which $\forall a \in \mathbb{N}^*$, $n|a^n - a$)). Next in importance is that 1729 is the sixth pseudo prime in base 2 (number n not prime such that $n|2^{n-1} - 1$). Its third property is that it belongs among the terms of a simple generative series. The property expounded by Ramanujan from his hospital bed appears as the fourth principle. In reviewing the responses from the encyclopedia, one finds further that:

- 1729 is the thirteenth number of the form $n^3 + 1$;
- 1729 is the fourth "factorial sextuple", that is to say, a product of successive terms of the form $6n + 1$: $1729 = 1 \times 7 \times 13 \times 19$;
- 1729 is the ninth number of the form $n^3 + (n + 1)^3$;
- 1729 is the sum of the factors of a perfect square (33^2);
- 1729 is a number whose digits, when added together yield its largest factor ($1 + 7 + 2 + 9 = 19$ and $1729 = 7 \times 13 \times 19$);
- 1729 is the product of 19 a prime number, multiplied by 91, its inverse;
- 1729 is the total number of ways to express 33 as the sum of 6 integers.

The sequence encyclopedia of Neil Sloane comprises more than 150 000 sequences. A partial version retaining only the most important sequences of the database was published by Neil Sloane and Simon Plouffe[11] in 1995. It records a selection of 5487 sequences[11] and echoes an earlier publication by Sloane [9].

Approximately forty mathematicians constitute the "editorial committee" of the database, but any user may propose sequences. If approved, they are added to the database according to criteria of mathematical interest. Neil Sloane's flexibility is apparent in the ease with which he adds new sequences as they are proposed. A degree of filtering is inevitable to maintain the quality of the database. Further, there exist a large number of infinite families of sequences (all the sequences of the form (kn), all the sequences of the form (k^n), etc.), of which it is understood that only the first numbers are recorded in the encyclopedia. A program is also used in the event of a failure

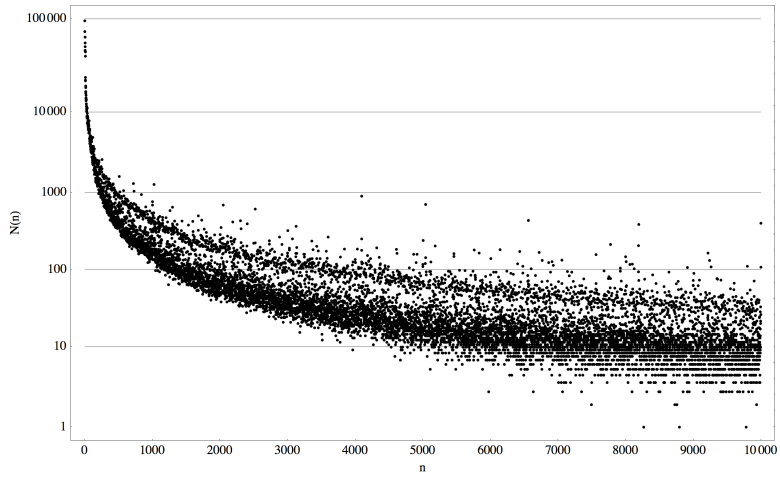

Figure 6.1: Number of occurrences of $N(n)$ as a function of n per n ranging from 1 to 10 000. Logarithmic scale in ordinate.

of a direct query which allows sequences of families that are not explicitly recorded in the encyclopedia to be recognized.

Each sequence recorded in the database appears in the form of its first terms. The size of first terms associated with each sequence is limited to approximately 180 digits. As a result, even if the sequence is easy to calculate, only its first terms will be expressed. Next to the first terms and extending from the beginning of the sequence, the encyclopedia proposes all sorts of other data about the sequence, e.g., the definitions of it and bibliographical references.

Sloane's integer encyclopedia is available in the form of a data file that is easy to read, and that contains only the terms retained for each sequence. One can download the data file free of charge and use it–with mathematical software, for example–to study the expressed numbers and conduct statistical research about the givens it contains.

One can, for example, ask the question: "Which numbers do not appear in Sloane's encyclopedia?" At the time of an initial calculation conducted in August 2008 by Philippe Guglielmetti, the smallest absent number tracked down was 8795, followed in order by 9935, 11147, 11446, 11612, 11630,... When the same calculation was made again in February 2009, the encyclopedia having been augmented by the addition of several hundreds of new sequences, the series of absent numbers was found to comprise 11630, 12067, 12407, 12887, 13258...

The instability over time of the sequence of missing numbers in the OEIS

suggests the need for a study of the distribution of numbers rather than of their mere presence or absence. Let us consider the number of properties of an integer, $N(n)$, while measuring it by the number of times n appears in the number file of the Sloane encyclopedia. The sequence $N(n)$ is certainly unstable over time, but it varies slowly, and certain ideas that one can derive from the values of $N(n)$ are nevertheless quite stable. The values of $N(n)$ are represented in Figure 1. In this logarithmic scale graph a cloud formation with regular decline curve is shown.

Let us give a few examples: the value of $N(1729)$ is 380 (February 2009), which is fairly high for a number of this order of magnitude. For its predecessor, one nevertheless calculates $N(1728) = 622$, which is better still. The number 1728 would thus have been easier for Ramanujan! Conversely, $N(1730) = 106$ and thus 1730 would have required a more elaborate answer than 1729.

The sequence $(N(n))_{n\in\mathbb{N}^*}$ is generally characterized by a decreasing curve. However, certain numbers n contradict this rule and possess more properties than their predecessors: $N(n) > N(n-1)$.

We can designate such numbers as "interesting". The first interesting number according to this definition is 15, because $N(15) = 34\,183$ and $N(14) = 32\,487$. Appearing next in order are 16, 23, 24, 27, 28, 29, 30, 35, 36, 40, 42, 45, 47, 48, 52, 53, etc.

We insist on the fact that, although unquestionably dependent on certain individual decisions made by those who participate in building the sequence database, the database is not in itself arbitrary. The number of contributors is very large, and the idea that the database represents an objective view (or at least an intersubjective view) of the numeric world could be defended on the grounds that it comprises the independent view of each person who contributes to it and reflects a stable mathematical (or cultural) reality.

Indirect support for the idea that the encyclopedia is not arbitrary, based as it is on the cumulative work of the mathematical community, is the general cloud-shaped formation of points determined by $N(n)$, which aggregates along a regular curve (see below).

Philippe Guglielmetti has observed that this cloud possesses a remarkable characteristic [2]: it is divided into two parts separated by a clear zone, as if the numbers sorted themselves into two categories, the more interesting above the clear zone, and the less interesting below the clear zone. We have given the name "Sloane's Gap" to the clear zone that divides in two the cloud

2. Personal communication with one of the authors, 16th of February, 2009.

representing the graph of the function $n \longmapsto N(n)$. Our goal in this paper is to describe the form of the cloud, and then to formulate an explanatory hypothesis for it.

6.3 Description of the cloud

Having briefly described the general form of the cloud, we shall direct ourselves more particularly to the gap, and we will investigate what characterizes the points that are situated above it.

6.3.1 General shape

The number of occurrences N is close to a grossly decreasing convex function of n, as one can see from Figure 1.

A logarithmic regression provides a more precise idea of the form of the cloud for n varying from 1 to 10 000. In this interval, the coefficient of determination of the logarithmic regression of $\ln(N(n))$ in n is of $r^2 = .81$, and the equation of regression gives the estimation:

$$\ln(N(n)) \simeq -1.33 \ln(n) + 14.76$$

or

$$\hat{N}(n) = \frac{k}{n^{1.33}},$$

where k is a constant having the approximate value 2.57×10^8, and $\hat{N}$ is the estimated value for N.

Thus the form of the function N is determined by the equation above. Is the existence of Sloane's gap natural then, or does it demand a specific explanation? We note that to our knowledge, only one publication mentions the existence of this split [4].

6.3.2 Defining the gap

In order to study the gap, the first step is to determine a criterion for classification of the points. Given that the "gap" is not clearly visible for the first values of n, we exclude from our study numbers less than 300.

One empirical method of determining the boundary of the gap is the following: for the values ranging from 301 to 499, we use a straight line

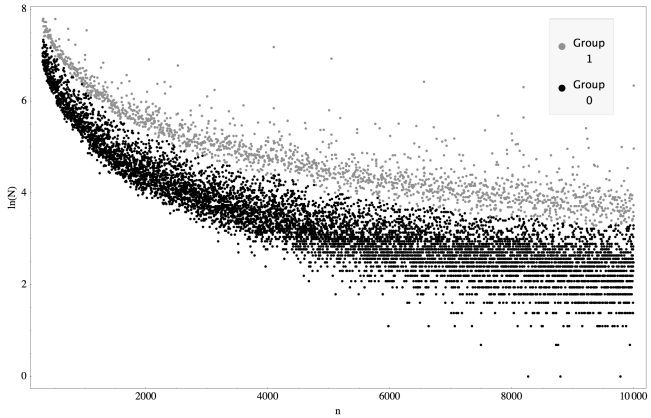

Figure 6.2: The curve represents the logarithmic regression of $\ln(N)$ as a function of n for n varying from 1 to 10 000. The grey scale points are those that are classified as being "above" the gap, while the others are classified as being "below" it.

adjusted "by sight", starting from the representation of $\ln(N)$ in functions of n. For subsequent values, we take as limit value of n the 82nd percentile of the interval $[n-c, n+c]$. c is fixed at 100 up to $n = 1000$, then to 350. It is clearly a matter of a purely empirical choice that does not require the force of a demonstration. The result corresponds roughly to what we perceive as the gap, with the understanding that a zone of uncertainty will always exist, since the gap is not entirely devoid of points. Figure 2 shows the resulting image.

6.3.3 Characteristics of numbers "above"

We will henceforth designate as A the set of abscissae of points classified "above" the gap by the method that we have used. Of the numbers between 301 and 10 000, 18.2% are found in A– 1767 values.

In this section, we are looking for the properties of these numbers. Philippe Guglielmetti has already remarked that the prime numbers and the powers of two seem to situate themselves more frequently above the gap. The idea is that certain classes of numbers that are particularly simple or of particular interest to the mathematician are over-represented.

Squares

83 square numbers are found between 301 and 10 000. Among these, 79 are located above the gap, and 4 below the gap, namely, numbers 361, 484, 529, and 676. Although they may not be elements of A, these numbers are close to the boundary. One can verify that they collectively realize the local maximums for $\ln(N)$ in the set of numbers classified under the cloud. One has, for example, $N(361) = 1376$, which is the local maximum of $\{N(n), n \in [325, 10\,000] \setminus A\}$. For each of these four numbers, Table 1 gives the number of occurrences N in Sloane's list, as well as the value limit that they would have to attain to belong to A.

95.2% of squares are found in A, as opposed to 17.6% of non-squares. The probability that a square number will be in A is thus 5.4 times greater than that for the other numbers.

n	$N(n)$	value limit
361	1376	1481
484	976	1225
529	962	1065
676	706	855

Table 1–List of the square numbers n found between 301 and 10 000 not belonging to A, together with their frequency of occurrence and the value of $N(n)$ needed for n to be classified in A.

Prime numbers

The interval under consideration contains 1167 prime numbers. Among them, 3 are not in A: the numbers 947, 8963, and 9623. These three numbers are very close to the boundary. 947 appears 583 times, while the limit of A is 584. Numbers 8963 and 6923 appear 27 times each, and the common limit is 28.

99.7% of prime numbers belong to A, and 92.9% of non-prime numbers belong to the complement of A. The probability that a prime number will belong to A is thus 14 times greater than the same probability for a non-prime number.

A multitude of factors

Another class of numbers that is seemingly over-represented in set A is the set of integers that have "a multitude of factors". This is based on the observation that the probability of belonging to A increases with the number of prime factors (counted with their multiples), as can be seen in Figure 3. To refine this idea we have selected the numbers n of which the number of prime factors (with their multiplicty) exceeds the 95th percentile, corresponding to the interval $[n-100, n+100]$.

811 numbers meet this criterion. Of these, 39% are found in A, as opposed to 16.3% for the other numbers. The probability that a number that has a multitude of prime factors will belong to A is thus 2.4 times greater than the same probability for a number that has a smaller number of factors. Table 2 shows the composition of A as a function of the classes that we have considered.

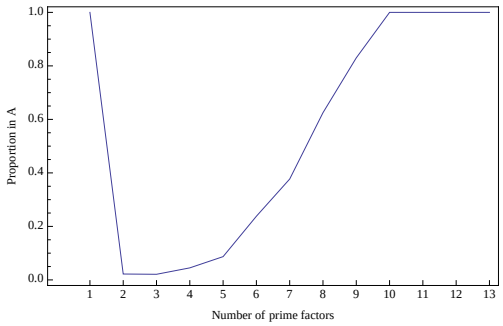

Figure 6.3: For each number of prime factors (counted with their multiples) one presents, the proportion of integers belonging to A is given. For the interval determined above, all numbers with at least 10 factors are in A.

class	number in A	% of A	% (cumulated)
primes	1164	65.9	65.9
squares	79	4.5	70.4
many factors	316	17.9	87.9

Table 2–For each class of numbers discussed above, they give the number of occurrences in A, the corresponding percentage and the cumulative percentage in A.

Other cases

The set A thus contains almost all prime numbers, 95% of squares, and a significant percentage of numbers that have a multitude of factors and all the numbers possessing at least ten prime factors (counted with their multiplicity).

These different classes of numbers by themselves represent 87.9% of A. Among the remaining numbers, some evince outstanding properties, for example, linked to decimal notation, as in: 1111, 2222, 3333.... Others have a simple form, such as 1023, 1025, 2047, 2049... that are written $2^n + 1$ or $2^n - 1$.

When these cases that for one reason or another possess an evident "simplicity" are eliminated, there remains a proportion of less than 10% of numbers in A for which one cannot immediately discern any particular property.

6.4 Explanation of the cloud-shape formation

6.4.1 Overview of the theory of algorithmic complexity

Save in a few exceptional cases, for a number to possess a multitude of properties implies that the said properties are simple, where simple is taken to mean "what may be expressed in a few words". Conversely, if a number possesses a simple property, then it will possess many properties. For example, if n is a multiple of 3, then n will be a even multiple of 3 or a odd multiple of 3. Being a "even multiple of 3" or "odd multiple of 3" is a little more complex than just being a "multiple of 3", but it is still simple enough, and one may further propose that many sequences in Sloane's database are actually sub-sequences of other, simpler ones. In specifying a simple property, its definition becomes more complex (by generating a subsequence of itself), but since there are many ways to specify a simple property, any number that possesses a simple property necessarily possesses numerous properties that are also simple.

The property of n corresponding to a high value of $N(n)$ thus seems related to the property of admitting a "simple" description. The value $N(n)$ appears in this context as an indirect measure of the simplicity of n, if one designates as "simple" the numbers that have properties expressible in a few words.

Algorithmic complexity theory[6, 2, 7] assigns a specific mathematical sense to the notion of simplicity, as the objects that "can be described with a short definition". Its modern formulation can be found in the work of Li and Vitanyi[8], and Calude[1].

Briefly, this theory proposes to measure the complexity of a finite object in binary code (for example, a number written in binary notation) by the length of the shortest program that generates a representation of it. The reference to a universal programming language (insofar as all computable functions can possess a program) leads to a theorem of invariance that warrants a certain independence of the programming language.

More precisely, if L_1 and L_2 are two universal languages, and if one notes K_{L_1} (resp. K_{L_2}) algorithmic complexity defined with reference to L_1 (resp. to L_2), then there exists a constant c such that $|K_{L_1}(s) - K_{L_2}(s)| < c$ for all finite binary sequences s.

A theorem (see for example [theorem 4.3.3. page 253 in [8]]) links the probability of obtaining an object s (by activating a certain type of universal TM–called optimal–running on binary input where the bits are chosen uniformly random) and its complexity $K(s)$. The rationale of this theorem is that if a number has many properties then it also has a simple property.

The translation of this theorem for $N(n)$ is that if one established a universal language L, and established a complexity limit M (only admitting descriptions of numbers capable of expression in fewer than M symbols), and counted the number of descriptions of each integer, one would find that $\frac{N(n)}{M}$ (where $M = \sum_{i\in\mathbb{N}} N(i)$) is approximately proportional to: $\frac{1}{2^{K(n)}}$:

$$\frac{N(n)}{M} = \frac{1}{2^{K(n)+O(\ln(\ln(n)))}}.$$

Given that $K(n)$ is non computable because of the undecidability of the halting problem and the role of the additive constants involved, a precise calculation of the expected value of $N(n)$ is impossible. By contrast, the strong analogy between the theoretical situation envisaged by algorithmic complexity and the situation one finds when one examines $N(n)$ inferred from Sloane's database, leads one to think that $N(n)$ should be asymptotically dependent on $\frac{1}{2^{K(n)}}$. Certain properties of $K(n)$ are obliquely independent of the reference language chosen to define K. The most important of these are:

- $K(n) < \log_2(n) + 2\log_2(\log_2(n)) + c'$ (c' a constant)
- the proportion of n of a given length (when written in binary) for which $K(n)$ recedes from $\log_2(n)$ decreases exponentially (precisely speaking,

less than an integer among 2^q of length k, has an algorithmic complexity $K(n) \leq k - q$).

In graphic terms, these properties indicate that the cloud of points obtained from writing the following $\frac{1}{2^{K(n)}}$ would be situated above a curve defined by

$$f(n) \approx \frac{h}{2^{\log_2(n)}} = \frac{h}{n}$$

(h being a constant), and that all the points cluster on the curve, with the density of the points deviating from the curve decreasing rapidly.

This is indeed the situation we observe in examining the curve giving $N(n)$. The theory of algorithmic information thus provides a good description of what is observable from the curve $N(n)$. That justifies an a posteriori recourse to the theoretical concepts of algorithmic complexity in order to understand the form of the curve $N(n)$. By contrast, nothing in the theory leads one to expect a gap like the one actually observed. To the contrary, continuity of form is expected from the fact that $n + 1$ is never much more complex than n.

To summarize, if $N(n)$ represented an objective measure of the complexity of numbers (the larger $N(n)$ is, the simpler n), these values would then be comparable to those that yield $\frac{1}{2^{K(n)}}$. One should thus observe a rapid decrease in size, and a clustering of values near the base against an oblique curve, but one should not observe a gap, which presents itself here as an anomaly.

To confirm the conclusion that the presence of the gap results from special factors and render it more convincing, we have conducted a numerical experiment.

We define random functions f in the following manner (thanks to the algebraic system *Mathematica*):

1. Choose at random a number i between 1 and 5 (bearing in mind in the selection the proportions of functions for which $i = 1$, $i = 2$, ..., $i = 5$ among all those definable in this way).
2. If $i = 1$, f is defined by choosing uniformly at random a constant $k \in \{1, ..., 9\}$, a binary operator φ from among the following list: $+$, $\times$, and subtraction sign, in a uniform manner, and a unary operand g that is identity with probability .8, and the function squared with probability .2 (to reproduce the proportions observed in Sloane's database). One therefore posits $f_i(n) = \varphi(g(n), k)$.

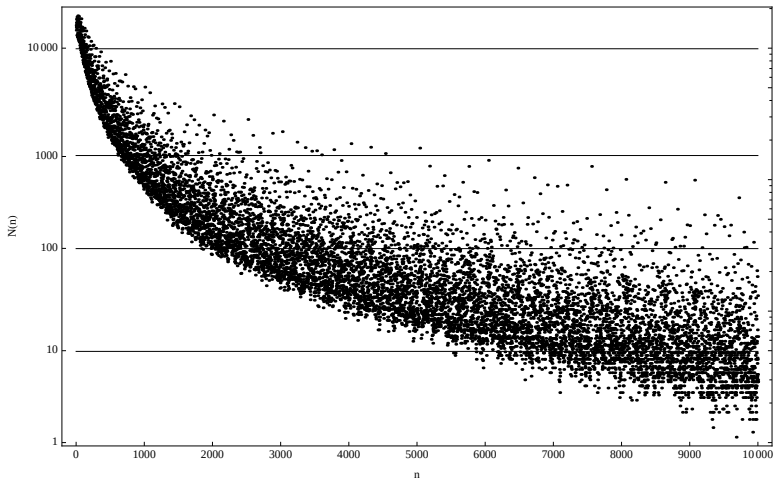

Figure 6.4: Graph of $N(n)$ obtained with random functions, similar to that belonging to Sloane's Database (Figure 1). Eight million values have been generated.

3. If $i \geq 2$, f_i is defined by $f_i(n) = \varphi(g(f_{i-1}(n)), k)$, where k is a random integer found between 1 and 9, g and φ are selected as described in the point 2 (above), and f_{i-1} is a random function selected in the same manner as in 2.

For each function f that is generated in this way, one calculates $f(n)$ for $n = 1,\ldots,20$. These terms are regrouped and counted as for $N(n)$. The results appear in Figure 4. The result confirms what the relationship with algorithmic complexity would lead us to expect. There is a decreasing oblique curve with a mean near 0, with clustering of the points near the base, but no gap.

6.4.2 The gap: A social effect?

This anomaly with respect to the theoretical implications and modeling is undoubtedly a sign that what one sees in Sloane's database is not a simple objective measure of complexity (or of intrinsic mathematical interest), but rather a trait of psychological or social origin that mars its pure expression. That is the hypothesis that we propose here. Under all circumstances, a purely mathematical vision based on algorithmic complexity would encounter an obstacle here, and the social hypothesis is both simple and natural owing to the fact that Sloane's database, while it is entirely "objective", is also a social construct.

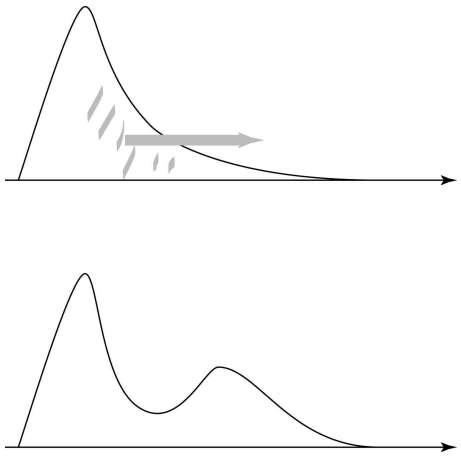

Figure 6.5: The top figure above represents the local distribution of N expected without taking into account the social factor.

Figure 5 illustrates and specifies our hypothesis that the mathematical community is particularly interested in certain numbers of moderate or weak complexity (in the central zone or on the right side of the distribution), and this interest creates a shift toward the right-hand side of one part of the distribution (schematized here by the grey arrow). The new distribution that develops out of it (represented in the bottom figure) presents a gap.

We suppose that the distribution anticipated by considerations of complexity is deformed by the social effect concomitant with it: mathematicians are more interested in certain numbers that are linked to selected properties by the scientific community. This interest can have cultural reasons or mathematical reasons (as per results already obtained), but in either case it brings with it an over-investment on the part of the mathematical community. The numbers that receive this specific over-investment are not in general complex, since interest is directed toward them because certain regularities have been discovered in them. Rather, these numbers are situated near the pinnacle of a theoretical asymmetrical distribution. Owing to the community's over-investment, they are found to have shifted towards the right-hand side of the distribution, thus explaining Sloane's gap.

It is, for example, what is generated by numbers of the form $2^n + 1$, all in A, because arithmetical results can be obtained from this type of number that are useful to prime numbers. Following some interesting preliminary discoveries, scientific investment in this class of integers has become intense, and they appear in numerous sequences. Certainly, $2^n + 1$ is objectively a simple number, and thus it is normal that it falls above the gap. Nevertheless,

the difference in complexity between $2^n + 1$ and $2^n + 2$ is weak. We suppose that the observed difference also reflects a social dynamic which tends to augment $N(2^n + 1)$ for reasons that complexity alone would not entirely explain.

6.5 Conclusion

The cloud of points representing the function N presents a general form evoking an underlying function characterized by rapid decrease and "clustering near the base" (local asymmetrical distribution). This form is explained, at least qualitatively, by the theory of algorithmic information.

If the general cloud formation was anticipated, the presence of Sloane's gap has, by contrast, proved more challenging to its observers. This gap has not, to our knowledge, been successfully explained on the basis of uniquely numerical considerations that are independent of human nature as it impinges on the work of mathematics. Algorithmic complexity anticipates a certain "continuity" of N, since the complexity of $n + 1$ is always close to that of n. The discontinuity that is manifest in Sloane's gap is thus difficult to attribute to purely mathematical properties independent of social contingencies.

By contrast, as we have seen, it is explained very well by the conduct of research that entails the over-representation of certain numbers of weak or medium complexity. Thus the cloud of points representing the function N shows features that can be understood as being the result of at the same time human and purely mathematical factors.

Bibliography

[1] CALUDE, C.S. *Information and Randomness: An Algorithmic Perspective.* (Texts in Theoretical Computer Science. An EATCS Series), Springer; 2nd. edition, 2002.

[2] CHAITIN, G.J. *Algorithmic Information Theory,* Cambridge University Press, 1987.

[3] CIPRA, B. *Mathematicians get an on-line fingerprint file,* Science, 205, (1994), p. 473.

[4] DELAHAYE, J.-P. *Mille collections de nombres,* Pour La Science, 379, (2009), p. 88-93.

[5] DELAHAYE, J.-P., ZENIL, H. "On the Kolmogorov-Chaitin complexity for short sequences", in CALUDE, C.S. (ed.) *Randomness and Complexity: from Chaitin to Leibniz*, World Scientific, p. 343-358, 2007.

[6] KOLMOGOROV, A.N. *Three approaches to the quantitative definition of information*. Problems of Information and Transmission, 1(1): 1–7, 1965.

[7] LEVIN, L. *Universal Search Problems.* 9(3): 265-266, 1973 (c). (submitted: 1972, reported in talks: 1971). English translation in: B.A.Trakhtenbrot. *A Survey of Russian Approaches to Perebor (Brute-force Search) Algorithms.* Annals of the History of Computing 6(4): 384-400, 1984.

[8] LI, M., VITANYI, P. *An introduction to Kolmogorov complexity and its applications,* Springer, 1997.

[9] SLOANE, N.J.A. *A Handbook of Integer Sequences,* Academic Press, 1973.

[10] SLOANE, N.J.A. *The on-line encyclopedia of integer sequences,* Notices of the American Mathematical Society, 8, (2003), p. 912-915.

[11] SLOANE, N.J.A. PLOUFFE, S. *The Encyclopedia of Integer Sequences,* Academic Press, 1995.

Quatrième partie

Réflexions

Chapitre 7

On the Algorithmic Nature of the World

Published in H. Zenil and J-P. Delahaye, "On the Algorithmic Nature of the World", in G. Dodig-Crnkovic and M. Burgin (eds), *Information and Computation, World Scientific, 2010.*

7.1 Introduction

We propose a test based on the theory of algorithmic complexity and an experimental evaluation of Levin's universal distribution to identify evidence in support of or in contravention of the claim that the world is algorithmic in nature. To this end we have undertaken a statistical comparison of the frequency distributions of data from physical sources on the one hand–repositories of information such as images, data stored in a hard drive, computer programs and DNA sequences–and the frequency distributions generated by purely algorithmic means on the other–by running abstract computing devices such as Turing machines, cellular automata and Post Tag systems. Statistical correlations were found and their significance measured.

A statistical comparison has been undertaken of the frequency distributions of data stored in physical repositories on the one hand–DNA sequences,

images, files in a hard drive–and of frequency distributions produced by purely algorithmic means on the other–by running abstract computational devices like Turing machines, cellular automata and Post Tag systems.

A standard statistical measure is adopted for this purpose. The Spearman rank correlation coefficient quantifies the strength of the relationship between two variables, without making any prior assumption as to the particular nature of the relationship between them.

7.1.1 Levin's universal distribution

Consider an unknown operation generating a binary string of length k bits. If the method is uniformly random, the probability of finding a particular string s is exactly 2^{-k}, the same as for any other string of length k. However, data is usually produced not at random but by a process. There is a measure which describes the expected output frequency distribution of an abstract machine running a program. A process that produces a string s with a program p when executed on a universal Turing machine T has probability $m(s)$. As p is itself a binary string, $m(s)$ can be defined as being the probability that the output of a universal prefix Turing machine T is s when provided with a sequence of fair coin flip inputs interpreted as a program. Formally,

$$m(s) = \Sigma_{T(p)=s} 2^{-|p|} \tag{7.1}$$

where the sum is over all halting programs p for which T outputs the string s, with $|p|$ the length of the program p. As T is a prefix universal Turing machine, the set of valid programs forms a prefix-free set [1] and thus the sum is bounded due to Kraft's inequality. For technical details see [1, 9, 3].

Formulated by Leonid Levin[8], m has many remarkable properties[7]. It is closely related to the concept of algorithmic complexity[2] in that the largest value of the sum of programs is dominated by the shortest one, so one can actually write $m(s)$ as follows:

$$m(s) = 2^{-K(s)+O(1)} \tag{7.2}$$

In a world of computable processes, $m(s)$ establishes that simple patterns which result from simple processes are likely, while complicated patterns

1. No element is a prefix of any other, a property necessary to keep $0 < m(s) < 1$ for all s and therefore a valid probability measure

produced by complicated processes (long programs) are relatively unlikely.

It is worth noting that, unlike other probability measures, m is not only a probability distribution establishing that there are some objects that have a certain probability of occurring according to said distribution, it is also a distribution specifying the order of the particular elements in terms of their individual algorithmic complexity.

7.2 The null hypothesis

When looking at a large-enough set of data following a distribution, one can in statistical terms safely assume that the source generating the data is of the nature that the distribution suggests. Such is the case when a set of data follows, for example, a Gaussian distribution, where depending on certain statistical variables, one can say with a high degree of certitude that the process generating the data is of a random nature.

When observing the world, the outcome of a physical phenomenon f can be seen as the result of a natural process P. One may ask how the probability distribution of a set of process of the type of P looks like.

If one would like to know whether the world is algorithmic in nature one would need first to tell how an algorithmic world would look like. To accomplish this, we've conceived and performed a series of experiments to produce data by purely algorithmic means in order to compare sets of data produced by several physical sources. At the right level a simplification of the data sets into binary language seems always possible. Each observation can measure one or more parameters (weight, location, etc.) of an enumeration of independent distinguishable values, a discrete sequence of values [2].

If there is no bias in the sampling method or the generating process itself and no information about the process is known, the principle of indifference[14] [3] states that if there are $n > 1$ possibilities mutually exclusive, collectively exhaustive and only distinguishable for their names then each possibility should be assigned a probability equal to $1/n$ as the simplest non-informative prior. The null hypothesis to test is that the frequency distributions studied herein from several different independent sources are closer

2. This might be seen as an oversimplification of the concept of a natural process and of its outcome when seen as a binary sequence, but the performance of a physical experiment always yields data written as a sequence of individual observations as a valid sample of certain phenomena.

3. also known as principle of insufficient reason.

to the experimental calculation of Levin's universal distribution than to the uniform (simplest non-informative prior) distribution. To this end average output frequency distributions by running abstract computing devices such as cellular automata, Post tag systems and Turing machines were produced on the one hand, and by collecting data to build distributions of the same type from the physical world on the other.

7.2.1 Frequency distributions

The distribution of a variable is a description of the relative number of times each possible outcome occurs in a number of trials. One of the most common probability distributions describing physical events is the normal distribution, also known as the Gaussian or Bell curve distribution, with values more likely to occur due to small random variations around a mean.

There is also a particular scientific interest in power-law distributions, partly from the ease with which certain general classes of mechanisms generate them. The demonstration of a power-law relation in some data can point to specific kinds of mechanisms that might underlie the natural phenomenon in question, and can indicate a connection with other, seemingly unrelated systems.

As explained however, when no information is available, the simplest distribution one can assume is the uniform distribution, in which values are equally likely to occur. In a macroscopic system at least, it must be assumed that the physical laws which govern the system are not known well enough to predict the outcome. If one does not have any reason to choose a specific distribution and no prior information is available, the uniform distribution is the one making no assumptions according to the principle of indifference. This is supposed to be the distribution of a balanced coin, an unbiased die or a casino roulette where the probability of an outcome k_i is $1/n$ if k_i can take one of n possible different outcomes.

7.2.2 Computing abstract machines

An abstract machine consists of a definition in terms of input, output, and the set of allowable operations used to turn the input into the output. They are of course algorithmic by nature (or by definition). Three of the most popular models of computation in the field of theoretical computer science were resorted to produce data of a purely algorithmic nature: these were

deterministic Turing machines (denoted by TM), one-dimensional cellular automata (denoted by CA) and Post Tag systems (TS).

The Turing machine model represents the basic framework underlying many concepts in computer science, including the definition of algorithmic complexity cited above. The cellular automaton is a well-known model which, together with the Post Tag system model, has been studied since the foundation of the field of abstract computation by some of its first pioneers. All three models are Turing-complete. The descriptions of the models follow formalisms used in [15].

Deterministic Turing machines

The Turing machine description consists of a list of rules (a finite program) capable of manipulating a linear list of cells, called the *tape*, using an access pointer called the *head*. The finite program can be in any one of a finite set of states Q numbered from 1 to n, with 1 the state at which the machine starts its computation. Each tape cell can contain 0 or 1 (there is no special blank symbol). Time is discrete and the steps are ordered from 0 to t with 0 the time at which the machine starts its computation. At any given time, the head is positioned over a particular cell and the finite program starts in the state 1. At time 0 all cells contain the same symbol, either 0 or 1. A rule i can be written in a 5-tuple notation as follows $\{s_i, k_i, s'_i, k'_i, d_i\}$, where s_i is the tape symbol the machine's head is scanning at time t, k_i the machine's current 'state' (instruction) at time t, s'_i a unique symbol to write (the machine can overwrite a 1 on a 0, a 0 on a 1, a 1 on a 1, or a 0 on a 0) at time $t+1$, k'_i a state to transition into (which may be the same as the one it was already in) at time $t+1$, and d_i a direction to move in time $t+1$, either to the right (R) cell or to the left (L) cell, after writing. Based on a set of rules of this type, usually called a transition table, a Turing machine can perform the following operations: 1. write an element from $A = \{0, 1\}$, 2. shift the head one cell to the left or right, 3. change the state of the finite program out of Q. When the machine is running it executes the above operations at the rate of one operation per step. At a time t the Turing machine produces an output described by the contiguous cells in the tape visited by the head.

Let $T(0), T(1), \ldots, T(n), \ldots$ be a natural recursive enumeration of all 2-symbol deterministic Turing machines. One can, for instance, begin enumerating by number of states, starting with all 2-state Turing machines, then 3-state, and so on. Let n, t and k be three integers. Let $s(T(n), t)$ be the part of the contiguous tape cells that the head visited after t steps. Let's consider all the k-tuples, i.e. all the substrings of length k

from $s(T(n),t) = \{s_1, s_2, \ldots, s_u\}$, i.e. the following $u-k+1$ k-tuples: $\{(s_1, \ldots, s_k), (s_2, \ldots, s_{k+1}), \ldots, (s_{u-k+1}, \ldots, s_u)\}$.

Now let N be a fixed integer. Let's consider the set of all the k-tuples produced by the first N Turing machines according to a recursive enumeration after running for t steps each. Let's take the count of each k-tuple produced.

From the count of all the k-tuples, listing all distinct strings together with their frequencies of occurrence, one gets a probability distribution over the finite set of strings in $\{0,1\}^k$.

For the Turing machines the experiments were carried out with 2-symbol 3-state Turing machines. There are $(4n)^{2n}$ possible different n-state 2-symbol Turing machines according to the 5-tuple rule description cited above. Therefore $(4 \times 3)^{(2\times 3)} = 2985984$ 2-symbol 3-state Turing machines. A sample of 2000 2-symbol 3-state Turing machines was taken. Each Turing machine's runtime was set to $t = 100$ steps starting with a tape filled with $0s$ and then once again with a tape filled with $1s$ in order to avoid any undesired asymmetry due to a particular initial set up.

One-dimensional Cellular Automata

An analogous standard description of one-dimensional 2-color cellular automata was followed. A one-dimensional cellular automaton is a collection of cells on a row that evolves through discrete time according to a set of rules based on the states of neighboring cells that are applied in parallel to each row over time. When the cellular automaton starts its computation, it applies the rules at a first step $t = 0$. If m is an integer, a neighborhood of m refers to the cells on both sides, together with the central cell, that the rule takes into consideration at row t to determine the value of a cell at the step $t+1$. If m is a fraction of the form p/q, then $p-1$ are the cells to the left and $q-1$ the cells to the right taken into consideration by the rules of the cellular automaton.

For cellular automata, the experiments were carried out with 3/2-range neighbor cellular automata starting from a single 1 on a background of $0s$ and then again starting from a single 0 on a background of $1s$ to avoid any undesired asymmetry from the initial set up. There are 2^{2m+1} possible states for the cells neighboring a given cell (m at each side plus the central cell), and two possible outcomes for the new cell; there are therefore a total of $2^{2^{2m+1}}$ one-dimensional m-neighbor 2-color cellular automata, hence $2^{2^{(2\times 3/2)+1}} = 65536$ cellular automata with rules taking two neighbors to the left and one

to the right. A sample of 2000 3/2-range neighbor cellular automata was taken.

As for Turing machines, let $A(1), A(2), \ldots, A(n), \ldots$ be a natural recursive enumeration of one dimensional 2-color cellular automata. For example, one can start enumerating them by neighborhood starting from range 1 (nearest-neighbor) to 3/2-neighbor, to 2-neighbor and so on, but this is not mandatory.

Let n, t and k be three integers. For each cellular automaton $A(n)$, let $s(A(n), t)$ denote the output of the cellular automata defined as the contiguous cells of the last row produced after $t = 100$ steps starting from a single black or white cell as described above, up to the length that the scope of the application of the rules starting from the initial configuration may have reached (usually the last row of the characteristic cone produced by a cellular automaton). As was done for Turing machines, tuples of length k were extracted from the output.

Post Tag Systems

A Tag system is a triplet (m, A, P), where m is a positive integer, called the deletion number. A is the alphabet of symbols (in this paper a binary alphabet). Finite (possibly empty) strings can be made of the alphabet A. A computation by a Tag system is a finite sequence of strings produced by iterating a transformation, starting with an initially given initial string at time $t = 0$. At each iteration m elements are removed from the beginning of the sequence and a set of elements determined by the production rule P is appended onto the end, based on the elements that were removed from the beginning. Since there is no generalized standard enumeration of Tag systems [4], a random set of rules was generated, each rule having equal probability. Rules are bound by the number of r elements (digits) on the left and right hand blocks of the rule. There are a total of $(k^r + 1 - 1)/(k - 1)^{k^n}$ possible rules if blocks up to length r can be added at each step. For $r = 3$, there are 50625 different 2-symbol Tag systems with deletion number 2. In this experiment, a sample of 2000 2-Tag systems (Tag systems with deletion number 2) were used to generate the frequency distributions of Tag systems to compare with.

An example of a rule is $\{0 \to 10, 1 \to 011, 00 \to \epsilon, 01 \to 10, 10 \to 11\}$, where no term on any side has more than 3 digits and there is no fixed number of elements other than that imposed to avoid multiple assignations of a string to several different, i.e. ambiguous, rules. The empty string ϵ can

4. To the authors' knowledge.

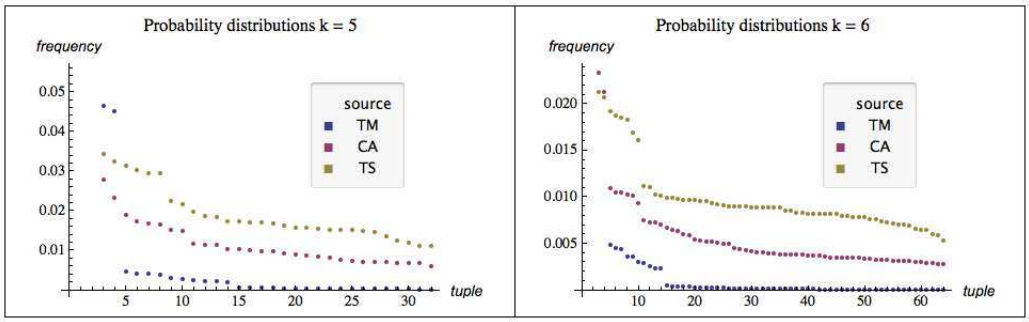

Figure 7.1: The output frequency distributions from running abstract computing machines. The x-axis shows all the 2^k tuples of length k sorted from most to least frequent. The y-axis shows the frequency values (probability between 0 and 1) of each tuple on the x-axis.

only occur among the right hand terms of the rules. The random generation of a set of rules yields results equivalent to those obtained by following and exhausting a natural recursive enumeration.

As an illustration[5], assume that the output of the first 4 Turing machines following an enumeration yields the output strings 01010, 11111, 11111 and 01 after running $t = 100$ steps. If $k = 3$, the tuples of length 3 from the output of these Turing machines are: $((010, 101, 010), (111, 111, 111), (111, 111, 111))$; or grouped and sorted from higher to lower frequency: $(111, 010, 101)$ with frequency values 6, 2, and 1 respectively. The frequency distribution is therefore $((111, 2/3), (010, 2/9), (101, 1/9))$, i.e. the string followed by the count divided by the total. If the strings have the same frequency value they are lexicographically sorted.

The output frequency distributions produced by abstract machines as described above are evidently algorithmic by nature (or by definition), and they will be used both to be compared one to each other, and to the distributions extracted from the physical real world.

7.2.3 Physical sources

Samples from physical sources such as DNA sequences, random images from the web and data stored in a hard drive were taken and transformed

5. For illustration only no actual enumeration was followed.

into data of the same type (i.e. binary tuples) of the produced by the abstract computing machines. We proceeded as follows: a representative set of random images was taken from the web using the random image function available online at the Wikipedia Creative Commons website at *http://commons.wikimedia.org/wiki/Special:Random/File* (as of May, 2009), none of them larger than 1500 linear pixels[6] to avoid any bias due to very large images. All images were transformed using the *Mathematica* function *Binarize* that converts multichannel and color images into black-and-white images by replacing all values above a globally determined threshold. Then all the k-tuples for each row of the image were taken and counted, just as if they had been produced by the abstract machines from the previous section.

Another source of physical information for comparative purposes was a random selection of human gene sequences of Deoxyribonucleic acid (or simply DNA). The DNA was extracted from a random sample of 100 different genes (the actual selection is posted in the project website cited in section 8.9).

There are four possible encodings for translating a DNA sequence into a binary string using a single bit for each letter: $\{G \to 1,\ T \to 1,\ C \to 0,\ A \to 0\}$, $\{G \to 0,\ T \to 1,\ C \to 0,\ A \to 1\}$, $\{G \to 1,\ T \to 0,\ C \to 1,\ A \to 0\}$, $\{G \to 0,\ T \to 0,\ C \to 1,\ A \to 1\}$.

To avoid any artificially induced asymmetries due to the choice of a particular encoding, all four encodings were applied to the same sample to build a joint frequency distribution. All the k-tuples were counted and ranked likewise.

There might be still some biases by sampling genes rather than sampling DNA segments because genes might be seen as conceived by researchers to focus on functional segments of the DNA. We've done however the same experiments taking only a sample of a sample of genes, which is not a gene by itself but a legitimate sample of the DNA, producing the same results (i.e. the distributions remain stable). Yet, the finding of a higher *algorithmicity* when taking gene samples as opposed to DNA general sampling might suggest that effectively there is an embedded encoding for genes in the DNA as functional subprograms in it, and not a mere research convenience.

A third source of information from the real world was a sample of data contained in a hard drive. A list of all the files contained in the hard drive was generated using a script, and a sample of 100 files was taken for comparison, with none of the files being greater than 1 Mb in order to avoid any bias due

6. The sum of the width and height.

to a very large file. The stream was likewise cut into k-tuples, counted and ranked to produce the frequency distribution, as for DNA. The count of each of the sources yielded a frequency distribution of k-tuples (the binary strings of length k) to compare with.

One may think that data stored in a hard drive already has a strong algorithmic component by the way that it has been produced (or stored in a digital computer) and therefore it makes no or less sense to compare with to any of the algorithmic or empirical distributions. It is true that the data stored in a hard drive is in the middle of what we may consider the abstract and the physical worlds, which makes it however interesting as an experiment by its own from our point of view. But more important, data stored in a hard drive is of very different nature, from text files subject to the rules of language, to executable programs, to music and video, all together in a single repository. Hence, it is not obvious at all why a frequency distribution from such a rich source of different kind of data might end up resembling to other distributions produced by other physical sources or by abstract machines.

7.2.4 Hypothesis testing

The frequency distributions generated by the different sources were statistically compared to look for any possible correlation. A correlation test was carried out and its significance measured to validate either the null hypothesis or the alternative (the latter being that the similarities are due to chance).

Each frequency distribution is the result of the count of the number of occurrences of the k-tuples from which the binary strings of length k were extracted. Comparisons were made with k set from 4 to 7.

Spearman's rank correlation coefficient

The Spearman rank correlation coefficient[13] is a non-parametric measure of correlation that makes no assumptions about the frequency distribution of the variables. Spearman's rank correlation coefficient is equivalent to the Pearson correlation on ranks. Spearman's rank correlation coefficient is usually denoted by the Greek letter ρ.

The Spearman rank correlation coefficient is calculated as follows:

$$\rho = 1 - ((6 \sum d_i^2)/(n(n^2 - 1))) \tag{7.3}$$

where d_i is the difference between each rank of corresponding values of x and y, and n the number of pairs of values.

Spearman's rank correlation coefficient can take real values from -1 to 1, where -1 is a perfect negative (inverse) correlation, 0 is no correlation and 1 is a perfect positive correlation.

The approach to testing whether an observed ρ value is significantly different from zero, considering the number of elements, is to calculate the probability that it would be greater than or equal to the observed ρ given the null hypothesis using a permutation test[6] to ascertain that the obtained value of ρ obtained is unlikely to occur by chance (the alternative hypothesis). The common convention is that if the value of ρ is between 0.01 and 0.001 the correlation is strong enough, indicating that the probability of having found the correlation is very unlikely to be a matter of chance, since it would occur one time out of hundred (if closer to 0.01) or a thousand (if closer to 0.001), while if it is between 0.10 and 0.01 the correlation is said to be weak, although yet quite unlikely to occur by chance, since it would occur one time out of ten (if closer to 0.10) or a hundred (if closer to 0.01) [7]. The lower the significance level, the stronger the evidence in favor of the null hypothesis. Tables 7.2.4, 7.2.4, 7.2.4 and 7.2.4 show the Spearman coefficients between all the distributions for a given tuple length k.

When graphically compared, the actual frequency values of each tuple among the 2^k unveil a correlation in values along different distributions. The x and y axes are in the same configuration as in the graph 8.3: The x-axis plots the 2^k tuples of length k but unlike the graph 8.3 they are lexicographically sorted (as the result of converting the binary string into a decimal number). The table 7.2.4 shows this lexicographical order as an illustration for $k = 4$. The y-axis plots the frequency value (probability between 0 and 1) for each tuple on the x-axis.

7.2.5 The problem of overfitting

When looking at a set of data following a distribution, one can safely claim in statistical terms that the source generating the data is of the nature that the distribution suggests. Such is the case when a set of data follows a model, where depending on certain variables, one can say with some degree of certitude that the process generating the data follows the model.

7. Useful tables with the calculation of levels of significance for different numbers of ranked elements are available online (e.g. at *http://www.york.ac.uk/depts/maths/histstat/tables/spearman.ps* as of May 2009).

CELLULAR AUTOMATA DISTRIBUTION

rank	*string (s)*	*count* ($pr(s)$)
1	1111	.35
2	0000	.34
3	1010	.033
4	0101	.032
5	0100	.026
6	0010	.026
7	0110	.025
8	1011	.024
9	1101	.023
10	1001	.023
11	0011	.017
12	0001	.017
13	1000	.017
14	1100	.016
15	0111	.016
16	1110	.017

HARD DRIVE DISTRIBUTION

rank	*string (s)*	*count* ($pr(s)$)
1	1111	.093
2	0000	.093
3	1110	.062
4	1000	.062
5	0111	.062
6	0001	.062
7	0100	.06
8	0010	.06
9	1101	.06
10	1011	.06
11	1100	.056
12	0011	.056
13	1001	.054
14	0110	.054
15	1010	.054
16	0101	.054

Table 7.1: Examples of frequency distributions of tuples of length $k = 4$, one from random files contained in a hard drive and another produced by running cellular automata. There are $2^4 = 16$ tuples each followed by its count (represented as a probability value between 0 and 1).

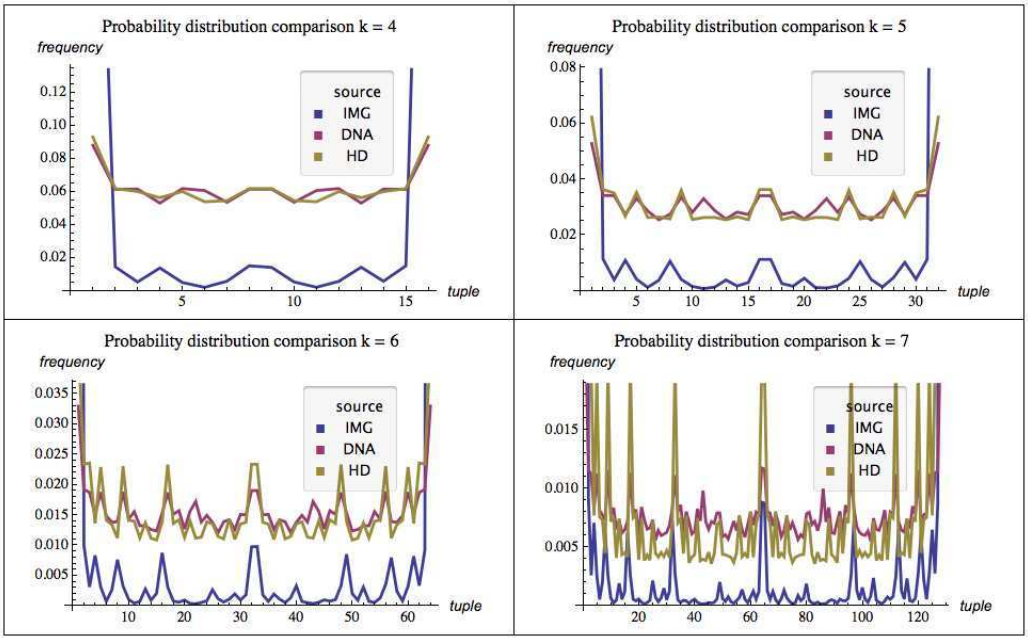

Figure 7.2: Frequency distributions of the tuples of length k from physical sources: binarized random files contained in a hard drive (HD), binarized sequences of Deoxyribonucleic acid (DNA) and binarized random images from the world wide web. The data points have been joined for clarity.

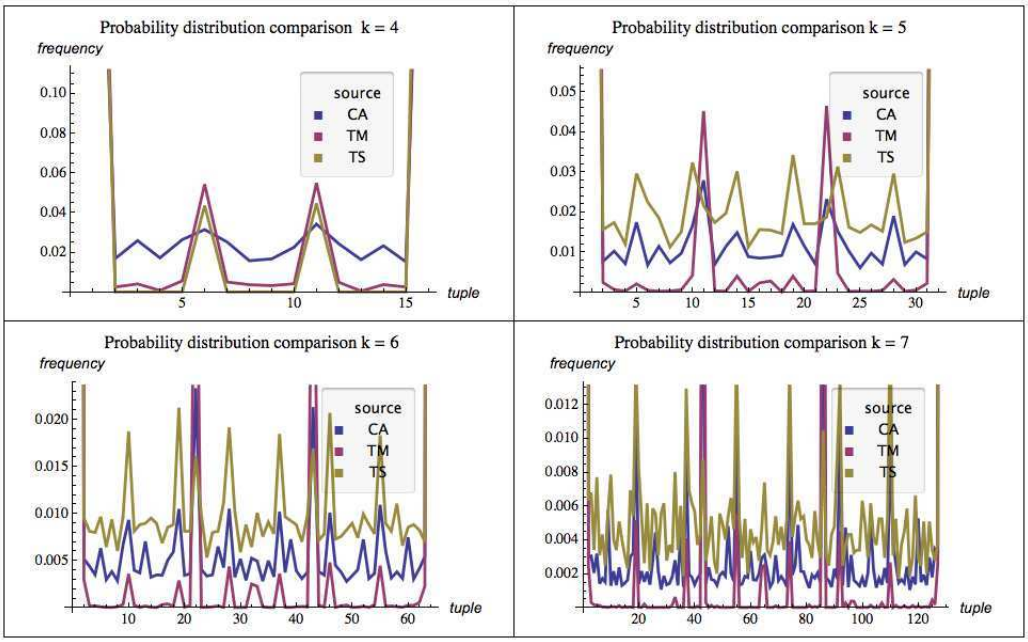

Figure 7.3: Frequency distributions of the tuples of length k from abstract computing machines: deterministic Turing machines (TM), one-dimensional cellular automata (CA) and Post Tag systems (TS).

Spearman coefficients for $K = 4$. Coefficients indicating a significant correlation are indicated by † while correlations with higher significance are indicated with ‡.

k = 4	HD	ADN	IMG	TM	CA	TS
HD	1‡	0.67‡	0.4	0.29	0.5	0.27
DNA	0.67‡	1‡	0.026	0.07	0.39†	0.52†
IMG	0.4†	0.026	1‡	0.31	0.044	0.24
TM	0.29	0.07	0.31	1‡	0.37†	0.044
CA	0.5†	0.39†	0.044	0.37	1‡	0.023
TS	0.27	0.52†	0.24	0.044	0.023	1‡

Spearman coefficients for $K = 5$.

k = 5	HD	ADN	IMG	TM	CA	TS
HD	1‡	0.62‡	0.09	0.31†	0.4‡	0.25†
ADN	0.62‡	1‡	0.30	0.11	0.39†	0.24†
IMG	0.09	0.30†	1‡	0.32†	0.60‡	0.10
TM	0.31†	0.11	0.32†	1‡	0.24†	0.07
CA	0.4‡	0.39†	0.24†	0.30†	1‡	0.18
TS	0.25†	0.24†	0.10	0.18	0.021	1‡

However, a common problem is the problem of over fitting, that is, a false model that may fit perfectly with an observed phenomenon [8]. Levin's universal distribution, however, is optimal over all distributions[7], in the sense that the algorithmic model is by itself the simplest possible model fitting the data if produced by some algorithmic process. This is because m is precisely the result of a distribution assuming the most simple model in algorithmic complexity terms, in which the shortest programs produce the elements leading the distribution. That doesn't mean, however, that it must necessarily be the right or the only possible model explaining the nature of the data, but the model itself is ill suited to an excess of parameters argument. A statistical comparison cannot actually be used to categorically prove or disprove a difference or similarity, only to favor one hypothesis over another.

7.3 Possible applications

Common data compressors are of the entropy coding type. Two of the most popular entropy coding schemes are the Huffman coding and the arith-

8. For example, Ptolemy's solar system model.

Spearman coefficients for $K = 6$.

k = 6	**HD**	**ADN**	**IMG**	**TM**	**CA**	**TS**
HD	1‡	0.58‡	0	0.27†	0.07	0.033
DNA	0.58‡	1‡	0	0.12	0.14	0
IMG	0	0	1‡	0.041	0.023	0.17†
TM	0.27†	0.12	0.041	1‡	0	0
CA	0.07	0.14	0.023	0	1‡	0.23†
TS	0.033	0	0.17†	0	0.23†	1‡

Spearman coefficients for $K = 7$.

k = 7	**HD**	**ADN**	**IMG**	**TM**	**CA**	**TS**
HD	1‡	0	0.091†	0.073	0	0.11†
DNA	0	1‡	0.07	0.028	0.12	0.019
IMG	0.091†	0.07	1‡	0.08†	0.15‡	0
TM	0.073	0.028	0.08†	1‡	0.03	0.039
CA	0	0.12	0.15‡	0.03	1‡	0
TS	0.11†	0.019	0	0.039	0	1‡

metic coding. Entropy coders encode a given set of symbols with the minimum number of bits required to represent them. These compression algorithms assign a unique prefix code to each unique symbol that occurs in the input, replacing each fixed-length input symbol by the corresponding variable-length prefix codeword. The length of each codeword is approximately proportional to the negative logarithm of the probability. Therefore, the most common symbols use the shortest codes.

Another popular compression technique based on the same principle is the run-length encoding (RLE) [9], wherein large runs of consecutive identical data values are replaced by a simple code with the data value and length of the run. This is an example of lossless data compression. However, none of these methods seem to follow any prior distribution [10], which means all of them are a posteriori techniques that after analyzing a particular image set their parameters to better compress it. A sort of prior compression distributions may be found in the so-called dictionary coders, also sometimes known as substitution coders, which operate by searching for matches between the text to be compressed and a set of strings contained in a static data structure.

In practice however, it is usually assumed that compressing an image

9. Implementations in different programming languages of the run-length encoding are available at *http://rosettacode.org/wiki/Run-length_encoding*

10. The authors were unable to find any reference to a general *prior* image compression distribution.

llustration of the simple lexicographical order of the 2^4 tuples of length $k = 4$ as plotted in the x-axis.

x-axis order	tuple	x-axis order	tuple
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

is image dependent, i.e. different from image to image. This is true when prior knowledge of the image is available, or there is enough time to spend in analyzing the file so that a different compression scheme can be set up and used every time. Effectively, compressors achieve greater rates because images have certain statistical properties which can be exploited. But what the experiments carried out here suggest for example is that a general optimal compressor for images based on the frequency distribution for images can be effectively devised and useful in cases when neither prior knowledge nor enough time to analyze the file is available. The distributions found, and tested to be stable could therefore be used for prior image compression techniques. The same sort of applications for other data sets can also be made, taking advantage of the kind of exhaustive calculations carried out in our experiments.

The procedure also may suggest a measure of *algorithmicity* relative to a model of computation: a system is more or less algorithmic in nature if it is more or less closer to the average distribution of an abstract model of computation. It has also been shown[4] that the calculation of these distributions constitute an effective procedure for the numerical evaluation of the algorithmic complexity of short strings, and a mean to provide stability to the definition–independent of additive constants–of algorithmic complexity.

7.4 The meaning of *algorithmic*

It may be objected that we have been careless in our use of the term *algorithmic*, not saying exactly what we mean by it. Nevertheless, *algorithmic* means nothing other than what this paper has tried to convey by the stance we have taken over the course of its arguments.

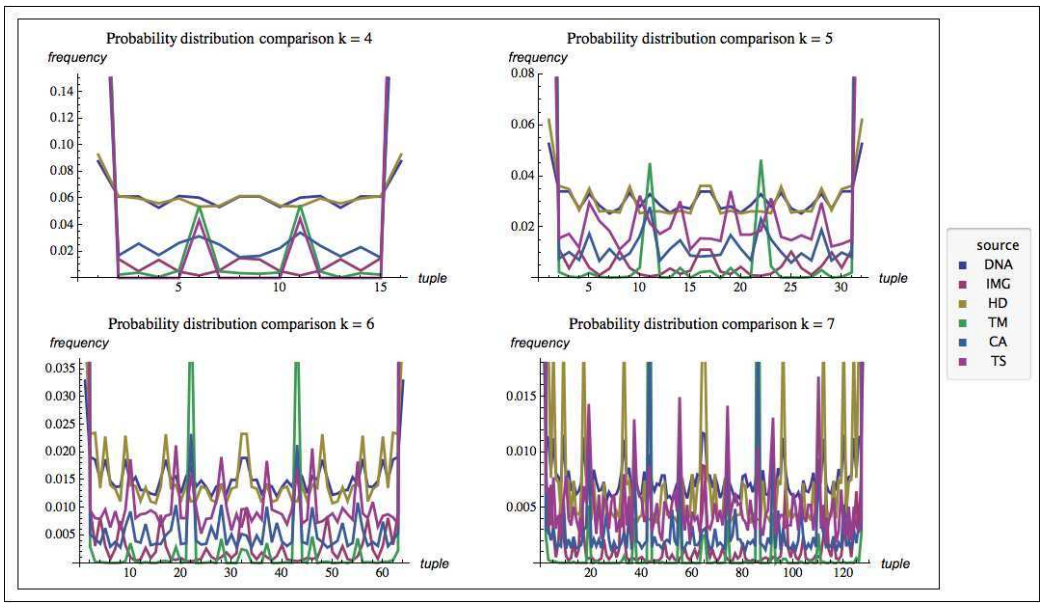

Figure 7.4: Comparisons of all frequency distributions of tuples of length k, from physical sources and from abstract computing machines.

In our context, *Algorithmic* is the adjective given to a set of processes or rules capable of being effectively carried out by a computer in opposition to a truly (indeterministic) random process (which is uncomputable). Classical models of computation[11] capture what an algorithm is but this paper (or what it implies) experimentally conveys the meaning of *algorithmic* both in theory and in practice, attempting to align the two. On the one hand, we had the theoretical basis of algorithmic probability. On the other hand we had the empirical data. We had no way to compare one with the other because of the non-computability of Levin's distribution (which would allow us to evaluate the algorithmic probability of an event). We proceeded, however, by constructing an experimental algorithmic distribution by running abstract computing machines (hence a purely algorithmic distribution), which we then compared to the distribution of empirical data, finding several kinds of correlations with different degrees of significance. For us therefore, algorithmic means the exponential accumulation of pattern producing rules and the isolation of randomness producing rules. In other words, the accumulation of simple rules.

Our definition of *algorithmic* is actually much stronger than the one di-

11. Albeit assuming the Church-Turing thesis.

rectly opposing true randomness. Because in our context something is algorithmic if it follows an algorithmic distribution (e.g. the experimental distribution we calculated). One can therefore take this to be a measure of *algorithmicity*: the degree to which a data set approaches an experimentally produced algorithmic distribution (assumed to be Levin's distribution). The closer it is to an algorithmic distribution the more algorithmic.

So when we state that a process is algorithmic in nature, we mean that it is composed by simple and deterministic rules, rules producing patterns, as algorithmic probability theoretically predicts. We think this is true of the market too, despite its particular dynamics, just as it is true of empirical data from other very different sources in the physical world that we have studied.

7.5 Conclusions

Our findings suggest that the information in the world might be the result of processes resembling processes carried out by computing machines. That does not necessarily imply a trivial reduction more than talking about algorithmic simple rules generating the data as opposed to random or truly complicated ones. Therefore we think that these correlations are mainly due to the following reason: that general physical processes are dominated by algorithmic simple rules. For example, processes involved in the replication and transmission of the DNA have been found[10] to be concatenation, union, reverse, complement, annealing and melting, all they very simple in nature. The same kind of simple rules may be the responsible of the rest of empirical data in spite of looking complicated or random. As opposed to simple rules one may think that nature might be performing processes represented by complicated mathematical functions, such as partial differential equations or all kind of sophisticated functions and possible algorithms. This suggests that the DNA carries a strong algorithmic component indicating that it has been developed as a result of algorithmic processes over the time, layer after layer of accumulated simple rules applied over and over.

So, if the distribution of a data set approaches a distribution produced by purely algorithmic machines rather than the uniform distribution, one may be persuaded within some degree of certainty, that the source of the data is of the same (algorithmic) nature just as one would accept a normal distribution as the footprint of a generating process of some random nature. The scenario described herein is the following: a collection of different distributions produced by different data sets produced by unrelated sources share some

properties captured in their frequency distributions, and a theory explaining the data (its regularities) has been presented in this paper.

There has hitherto been no way to either verify or refute the information-theoretic notion, beyond the metaphor, of whether the universe can be conceived as either the output of some computer program or as some sort of vast digital computation device as suggested by some authors[5, 12, 15, 11].

We think we've devised herein a valid statistical test independent of any bias toward either possibility. Some indications of correlations have been found having weak to strong significance. This is the case with distributions from the chosen abstract devices, as well as with data from the chosen physical sources. Each by itself turned out to show several degrees of correlation. While the correlation between the two sets was partial, each distribution was correlated with at least one distribution produced by an abstract model of computation. In other words, the physical world turned out to be statistically similar in these terms to the simulated one.

Bibliography

[1] C.S. Calude, *Information and Randomness: An Algorithmic Perspective (Texts in Theoretical Computer Science. An EATCS Series),* Springer, 2nd. edition, 2002.

[2] G.J. Chaitin, *Exploring Randomness,* Springer Verlag, 2001.

[3] R.G. Downey, D. Hirschfeldt, *Algorithmic Randomness and Complexity*, Springer Verlag, forthcoming, 2010.

[4] J.P. Delahaye, H. Zenil, On the Kolmogorov-Chaitin complexity for short sequences, in Cristian Calude (eds) *Complexity and Randomness: From Leibniz to Chaitin.* World Scientific, 2007.

[5] E. Fredkin, *Finite Nature,* available at http://www.digitalphilosophy.org/Home/, 1992.

[6] P.I. Good, *Permutation, Parametric and Bootstrap Tests of Hypotheses*, 3rd ed., Springer, 2005.

[7] W. Kirchherr, M. Li, *The miraculous universal distribution,* Mathematical Intelligencer , 1997.

[8] L. Levin, Universal Search Problems, 9(3):265-266, 1973 (c). (submitted: 1972, reported in talks: 1971). English translation in: B.A.Trakhtenbrot. *A Survey of Russian Approaches to Perebor (Brute-force Search) Algorithms.* Annals of the History of Computing 6(4):384-400, 1984.

[9] M. Li, P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications,* Springer, 3rd. Revised edition, 2008.

[10] Z. Li, *Algebraic properties of DNA operations,*, BioSystems, Vol.52, No.1-3, 1999.

[11] S. Lloyd, *Programming the Universe: A Quantum Computer Scientist Takes On the Cosmos,* Knopf, 2006.

[12] J. Schmidhuber, *Algorithmic Theories of Everything,* `arXiv:quant-ph/0011122v2`, 2000.

[13] W. Snedecor, WG. Cochran, *Statistical Methods*, Iowa State University Press; 8 edition, 1989.

[14] E. Thompson Jaynes, *Probability Theory: The Logic of Science,* Cambridge University Press, 2003.

[15] S. Wolfram, *A New Kind of Science,* Wolfram Media, Champaign, IL., 2002.

Chapitre 8

An Algorithmic Information-theoretic Approach to the Behavior of Financial Markets

Forthcoming in H. Zenil and J-P. Delahaye, *An Algorithmic Information-theoretic Approach to the Behavior of Financial Markets* themed issue on *"Nonlinearity, Complexity and Randomness"*, Journal of Economic Surveys, 2011.

8.1 Introduction

Using frequency distributions of daily closing price time series of several financial market indexes, we investigate whether the bias away from an equiprobable sequence distribution found in the data, predicted by algorithmic information theory, may account for some of the deviation of financial markets from log-normal, and if so for how much of said deviation and over what sequence lengths. We do so by comparing the distributions of binary sequences from actual time series of financial markets and series built up from purely algorithmic means. Our discussion is a starting point

for a further investigation of the market as a rule-based system with an *algorithmic* component, despite its apparent randomness, and the use of the theory of algorithmic probability with new tools that can be applied to the study of the market price phenomenon. The main discussion is cast in terms of assumptions common to areas of economics in agreement with an algorithmic view of the market.

8.2 Preliminaries

One of the main assumptions regarding price modeling for option pricing is that stock prices in the market behave as stochastic processes, that is, that price movements are log-normally distributed. Unlike classical probability, algorithmic probability theory has the distinct advantage that it can be used to calculate the likelihood of certain events occurring based on their information content. We investigate whether the theory of algorithmic information may account for some of the deviation from log-normal of the data of price movements accumulating in a power-law distribution.

We think that the power-law distribution may be an indicator of an information-content phenomenon underlying the market, and consequently that departures from log-normality can, given the accumulation of simple rule-based processes—-a manifestation of hidden structural complexity—-be accounted for by Levin's universal distribution, which is compatible with the distribution of the empirical data. If this is true, algorithmic probability could supply a powerful set of tools that can be applied to the study of market behavior. Levin's distribution reinforces what has been empirically observed, viz. that some events are more likely than others, that events are not independent of each other, and that their distribution depends on their information content. Levin's distribution is not a typical probability distribution inasmuch as it has internal structure placing the elements according to their structure specifying their exact place in the distribution, unlike other typical probability distributions that may indicate where some elements accumulate without specifying the particular elements themselves.

The methodological discipline of considering markets as algorithmic is one facet of the algorithmic approach to economics laid out in [32]. The focus is not exclusively on the institution of the market, but also on agents (of every sort), and on the behavioral underpinnings of agents (rational or otherwise) and markets (competitive or not, etc.).

We will show that the algorithmic view of the market as an alternative interpretation of the deviation from log-normal behavior of prices in financial markets is also compatible with some common assumptions in classical models of market behavior, with the added advantage that it points to the iteration of algorithmic processes as a possible cause of the discrepancies between the data and stochastic models.

We think that the study of frequency distributions and the application of algorithmic probability could constitute a tool for estimating and eventually understanding the information assimilation process in the market, making it possible to characterize the information content of prices.

The paper is organized as follows: In 8.3 a simplified overview of the basics of the stochastic approach to the behavior of financial markets is introduced, followed by a section discussing the apparent randomness of the market. In section 8.5, the theoretic-algorithmic approach we are proposing herein is presented, preceded by a short introduction to the theory of algorithmic information, and followed by a description of the hypothesis testing methodology 8.6.4. In 8.7.2, tables of frequency distributions of price direction sequences for five different stock markets are compared to equiprobable (normal independent) sequences of length 3 and 4 to length 10 and to the output frequency distributions produced by algorithmic means. The alternative hypothesis, that is that the market has an algorithmic component and that algorithmic probability may account for some of the deviation of price movements from log-normality is tested, followed by a backtesting section 8.7.2 before introducing further considerations in 8.8 regarding common assumptions in economics. The paper ends with a short section that summarizes conclusions and provides suggestions for further work in 8.9.

8.3 The traditional stochastic approach

When events are (random) independent of each other they accumulate in a normal (Gaussian) distribution. Stock price movements are for the most part considered to behave independently of each other. The random-walk like evolution of the stock market has motivated the use of Brownian motion for modeling price movements.

Brownian motion and financial modeling have been historically tied together[7], ever since Bachelier[2] proposed to model the price S_t of an asset on the Paris stock market in terms of a random process of Brownian motion W_t applied to the original price S_0. Thus $S_t = S_0 + \sigma W_t$.

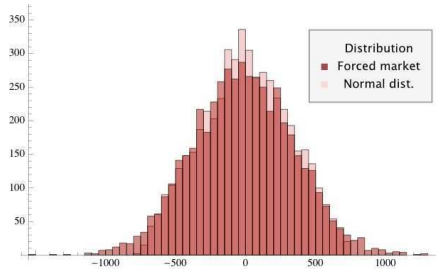

Figure 8.1: In a normal distribution any event is more or less like any other.

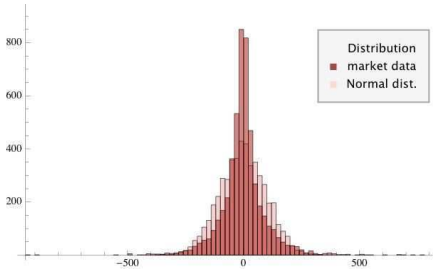

Figure 8.2: Events in a long-tailed (power-law) distribution indicate that certain days are not like any others.

The process S is sometimes called a *log* (or *geometric*) *Brownian motion*. Data of price changes from the actual markets are actually too *peaked* to be related to samples from normal populations. One can get a more convoluted model based on this process introducing or restricting the amount of randomness in the model so that it can be adjusted to some extent to account for some of the deviation of the empirical data to the supposedly log-normality.

A practical assumption in the study of financial markets is that the forces behind the market have a strong stochastic nature (see Figure 1 of a normal distribution and how a simulated market data may be forced to fit it). The idea stems from the main assumption that market fluctuations can be described by classical probability theory. The multiplicative version of Bachelier's model led to the commonly used Black-Scholes model, where the log-price S_t follows a random walk $S_t = S_0 exp[\sigma t + \sigma W_t]$.

The kind of distribution in which price changes accumulate is a power-law in which high-frequency events are followed by low-frequency events, with the short and very quick transition between them characterised by asymptotic behavior. Perturbations accumulate and are more frequent than if normally distributed, as happens in the actual market, where price movements accumulate in long-tailed distributions. Such a distribution often points to specific kinds of mechanisms, and can often indicate a deep connection with other, seemingly unrelated systems.

As found by Mandelbrot[24] in the 60's; prices do not follow a normal distribution; suggesting as it seems to be the case that some unexpected events happen more frequently than predicted by the Brownian motion model. On the right one walk was generated by taking the central column of a rule 30 cellular automaton (CA), another walk by using the `RandomInteger[]` random number function built in *Mathematica*. Only one is an actual sequence of price movements for 3 000 closing daily prices of the Dow Jones Index (DJI).

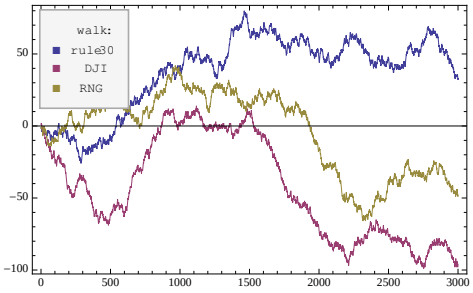

Figure 8.3: Simulated Brownian walks using a CA, a RNG and only one a true segment of daily closing prices of the DJI.

8.4 Apparent randomness in financial markets

The most obvious feature of essentially all financial markets is the apparent randomness with which prices tend to fluctuate. Nevertheless, the very idea of chance in financial markets clashes with our intuitive sense of the processes regulating the market. All processes involved seem deterministic. Traders do not only follow hunches but act in accordance with specific rules, and even when they do appear to act on intuition, their decisions are not random but instead follow from the best of their knowledge of the internal and external state of the market. For example, traders copy other traders, or take the same decisions that have previously worked, sometimes reacting against information and sometimes acting in accordance with it. Furthermore, nowadays a greater percentage of the trading volume is handled electronically, by computing systems (conveniently called algorithmic trading) rather than by humans. Computing systems are used for entering trading orders, for deciding on aspects of an order such as the timing, price and quantity, all of which cannot but be algorithmic by definition.

Algorithmic however, does not necessarily mean *predictable*. Several types of irreducibility, from non-computability to intractability to unpredictability, are entailed in most non-trivial questions about financial markets, as shown with clear examples in [32] and [33].

Wolfram's proposal for modeling market prices would have a simple program generate the randomness that occurs intrinsically. A plausible, if simple and idealised behavior is shown in the aggregate to produce intrinsically random behavior similar to that seen in price changes. In Figure 4, one can see that even in some of the simplest possible rule-based systems, structures emerge from a random-looking initial configuration with low information content. Trends and cycles are to be found amidst apparent randomness.

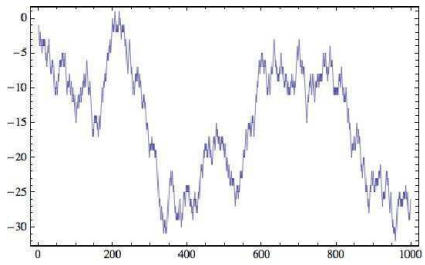

Figure 8.4: Patterns out of nothing: random walk by 1 000 data points generated using the *Mathematica* pseudo-random number generator based on a deterministic cellular automaton.

In [33] Wolfram asks whether the market generates its own randomness, starting from deterministic and purely algorithmic rules. Wolfram points out that the fact that apparent randomness seems to emerge even in very short timescales suggests that the randomness (or a source of it) that one sees in the market is likely to be the consequence of internal dynamics rather than of external factors. In economists' jargon, prices are determined by endogenous effects peculiar to the inner workings of the markets themselves, rather than (solely) by the exogenous effects of outside events.

Wolfram points out that pure speculation, where trading occurs without the possibility of any significant external input, often leads to situations in which prices tend to show more, rather than less, random-looking fluctuations. He also suggests that there is no better way to find the causes of this apparent randomness than by performing an almost step-by-step simulation, with little chance of beating the time it takes for the phenomenon to unfold—-the time scales of real world markets being simply too fast to beat. It is important to note that the intrinsic generation of complexity proves the stochastic notion to be a convenient assumption about the market, but not an inherent or essential one.

Economists may argue that the question is irrelevant for practical purposes. They are interested in decomposing time-series into a non-predictable and a presumably predictable signal in which they have an interest, what is traditionally called a trend. Whether one, both or none of the two signals is deterministic may be considered irrelevant as long as there is a part that

An example of a simple model of the market as shown in [33], where each cell of a cellular automaton corresponds to an entity buying or selling at each step. The behavior of a given cell is determined by the behavior of its two neighbors on the step before according to a rule. The plot on the left gives as a rough analog of a market price differences of the total numbers of black and white cells at successive steps. A rule like rule 90 is additive, hence reversible, which means that it does not destroy any information and has "memory" unlike the random walk model. Yet, due to its random looking behavior, it is not trivial shortcut the computation or foresee any successive step. There is some *randomness* in the initial condition of the cellular automaton rule that comes from outside the model, but the subsequent evolution of the system is fully deterministic. The way the series plot is calculated is written in *Mathematica* as follows `Accumulate[Total/@(CA/.{0 → −1})]` with *CA* the output evolution of rule 90 after 100 steps.

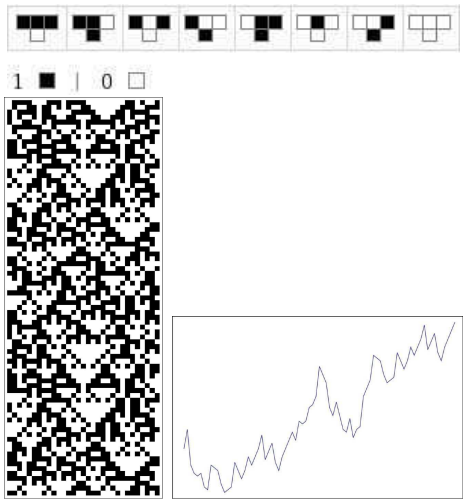

Figure 8.5: On the top, the rule 90 instruction table. On the left the evolution of rule 90 from a random init condition for 100 steps. On the bottom the total differences at every step between black and white cells.

is random-looking, hence most likely unpredictable and consequently worth leaving out.

What Wolfram's simplified model show, based on simple rules, is that despite being so simple and completely deterministic, these models are capable of generating great complexity and exhibit (the lack of) patterns similar to the apparent randomness found in the price movements phenomenon in financial markets. Whether one can get the kind of crashes in which financial markets seem to cyclicly fall into depends on whether the generating rule is capable of producing them from time to time. Economists dispute whether crashes reflect the intrinsic instability of the market, or whether they are triggered by external events. In a model in [19], for example, sudden large changes are internally generated suggesting large changes are more predictable—both in magnitude and in direction as the result of various interactions between agents. If Wolfram's intrinsic randomness is what leads the market one may think one could then easily predict its behavior if this were the case, but as suggested by Wolfram's Principle of Computational Equivalence it is reasonable to expect that the overall collective behavior of the market would look complicated to us, as if it were random, hence quite difficult to predict despite being or having a large deterministic component.

Wolfram's Principle of Computational Irreducibility[33] says that the only way to determine the answer to a computationally irreducible question is to perform the computation. According to Wolfram, it follows from his Principle of Computational Equivalence (PCE) that "*almost all processes that are not obviously simple can be viewed as computations of equivalent sophistication: when a system reaches a threshold of computational sophistication often reached by non-trivial systems, the system will be computationally irreducible.*"

8.5 An information-theoretic approach

From the point of view of cryptanalysis, the algorithmic view based on frequency analysis presented herein may be taken as a hacker approach to the financial market. While the goal is clearly to find a sort of password unveiling the rules governing the price changes, what we claim is that the password may not be immune to a frequency analysis attack, because it is not the result of a true random process but rather the consequence of the application of a set of (mostly simple) rules. Yet that doesn't mean one can crack the market once and for all, since for our system to find the said password it would have to outperform the unfolding processes affecting the

market—which, as Wolfram's PCE suggests, would require at least the same computational sophistication as the market itself, with at least one variable modeling the information being assimilated into prices by the market at any given moment. In other words, the market password is partially safe not because of the complexity of the password itself but because it reacts to the cracking method.

Whichever kind of financial instrument one looks at, the sequences of prices at successive times show some overall trends and varying amounts of apparent randomness. However, despite the fact that there is no contingent necessity of true randomness behind the market, it can certainly look that way to anyone ignoring the generative processes, anyone unable to see what other, non-random signals may be driving market movements.

von Mises' approach to the definition of a random sequence, which seemed at the time of its formulation to be quite problematic, contained some of the basics of the modern approach adopted by Per Martin-Löf[26]. It is during this time that the Keynesian[16] kind of induction may have been resorted to as a starting point for Solomonoff's seminal work[31] on algorithmic probability.

Martin-Löf gave the first suitable definition of a random sequence. Intuitively, an algorithmically random sequence (or random sequence) is an infinite sequence of binary digits that appears random to any algorithm. This contrasts with the idea of randomness in probability. In that theory, no particular element of a sample space can be said to be random. Martin-Löf randomness has since been shown to admit several equivalent characterizations in terms of compression, statistical tests, and gambling strategies.

The predictive aim of economics is actually profoundly related to the concept of predicting and betting. Imagine a random walk that goes up, down, left or right by one, with each step having the same probability. If the expected time at which the walk ends is finite, predicting that the expected stop position is equal to the initial position, it is called a martingale. This is because the chances of going up, down, left or right, are the same, so that one ends up close to one's starting position, if not exactly at that position. In economics, this can be translated into a trader's experience. The conditional expected assets of a trader are equal to his present assets if a sequence of events is truly random.

Schnorr[30, 8] provided another equivalent definition in terms of martingales. The martingale characterization of randomness says that no betting strategy implementable by any computer (even in the weak sense of constructive strategies, which are not necessarily computable) can make money

If market price differences accumulated in a normal distribution, a rounding would produce sequences of 0 differences only. The *mean* and the *standard deviation* of the market distribution are used to create a normal distribution, which is then subtracted from the market distribution. Rounding by the normal distribution cover, the elements in the tail are *extracted* as shown in Figure 6.

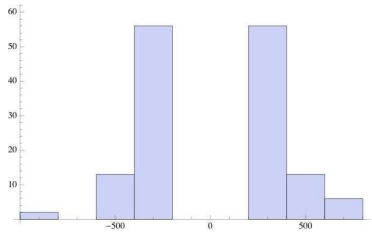

Figure 8.6: By extracting a normal distribution from the market distribution, the long-tail events are isolated.

betting on a random sequence. In a true random memoryless market, no betting strategy can improve the expected winnings, nor can any option cover the risks in the long term.

Over the last few decades, several systems have shifted towards ever greater levels of complexity and information density. The result has been a shift towards Paretian outcomes, particularly within any event that contains a high percentage of informational content [1].

Departures from normality could be accounted for by the algorithmic component acting in the market, as is consonant with some empirical observations and common assumptions in economics, such as rule-based markets and agents.

8.5.1 Algorithmic complexity

At the core of algorithmic information theory (AIT) is the concept of algorithmic complexity [2], a measure of the quantity of information contained in a string of digits. The algorithmic complexity of a string[18, 5] is defined as the length of the shortest algorithm that, when provided as input to a universal Turing machine or idealized simple computer, generates the string. A string has maximal algorithmic complexity if the shortest algorithm able to generate it is not significantly shorter than the string itself, perhaps allowing for a fixed additive constant. The difference in length between a string and

1. For example, if one plots the frequency rank of words contained in a large corpus of text data versus the number of occurrences or actual frequencies, Zipf showed that one obtains a power-law distribution.
2. Also known as program-size complexity, or Kolmogorov complexity.

the shortest algorithm able to generate it is the string's degree of compressibility. A string of low complexity is therefore highly compressible, as the information that it contains can be encoded in an algorithm much shorter than the string itself. By contrast, a string of maximal complexity is incompressible. Such a string constitutes its own shortest description: there is no more economical way of communicating the information that it contains than by transmitting the string in its entirety. In algorithmic information theory a string is algorithmically random if it is incompressible.

Algorithmic complexity is inversely related to the degree of regularity of a string. Any pattern in a string constitutes redundancy: it enables one portion of the string to be recovered from another, allowing a more concise description. Therefore highly regular strings have low algorithmic complexity, whereas strings that exhibit little or no pattern have high complexity.

The algorithmic complexity $K_U(s)$ of a string s with respect to a universal Turing machine U is defined as the binary length of the shortest program p that produces as output the string s.

$$K_U(s) = min\{|p|, U(p) = s)$$

Algorithmic complexity conveys the intuition that a random string should be incompressible: no program shorter than the size of the string produces the string.

Even though K is uncomputable as a function, meaning that there is no effective procedure (algorithm) to calculate it, one can use the theory of algorithmic probability to obtain exact evaluations of $K(s)$ for strings s short enough for which the halting problem can be solved for a finite number of cases due the size (and simplicity) of the Turing machines involved.

8.5.2 Algorithmic probability

What traders often end up doing in turbulent price periods is to leave aside the "any day is like any other normal day" rule, and fall back on their intuition, which leads to their unwittingly following a model we believe to be better fitted to reality and hence to be preferred at all times, not just in times of turbulence.

Intuition is based on weighting past experience, with experience that is closer in time being more relevant. This is very close to the concept of

algorithmic probability and the way it has been used (and was originally intended to be used[31]) in some academic circles as a theory of universal inductive inference[14].

Algorithmic probability assigns to objects an a priori probability that is in some sense universal[**?**]. This a priori distribution has theoretical applications in a number of areas, including inductive inference theory and the time complexity analysis of algorithms. Its main drawback is that it is not computable and thus can only be approximated in practice.

The concept of algorithmic probability was first developed by Solomonof[31] and formalized by Levin[20]. Consider an unknown process producing a binary string of length k bits. If the process is uniformly random, the probability of producing a particular string s is exactly 2^{-k}, the same as for any other string of length k. Intuitively, however, one feels that there should be a difference between a string that can be recognized and distinguished, and the vast majority of strings that are indistinguishable to us as regards whether the underlying process is truly random.

Assume one tosses a fair coin 20 three times and gets the following outcomes:

00000000000000000000
01100101110101001011
11101001100100101101

the first outcome would be very unlikely because one would expect a patternless outcome from a fair coin toss, one that resembles the second and third outcomes. In fact, it would be far more likely that a simple deterministic algorithmic process has generated this string. The same could be said for the market: one usually expects to see few if any patterns in its main indicators, mostly for the reasons set forth in section 8.4. Algorithmic complexity captures this expectation of patternlessness by defining what a random-looking string looks like. On the other hand, algorithmic probability predicts that random-looking outputs are the exception rather than the rule when the generating process is algorithmic.

There is a measure which describes the expected output of an abstract machine when running a random program. A process that produces a string s with a program p when executed on a universal Turing machine U has probability $m(s)$. As p is itself a binary string, $m(s)$ can be defined as being

the probability that the output of a universal Turing machine[3] U is s when provided with a sequence of fair coin flip inputs interpreted as a program.

$$m(s) = \Sigma_{U(p)=s} 2^{-|p|} = 2^{-K(s)+O(1)}$$

i.e. the sum over all the programs p for which the universal Turing machine U outputs the string s and halts.

Levin's universal distribution is so called because, despite being uncomputable, it has the remarkable property (proven by Leonid Levin himself) that among all the lower semi-computable semi-measures, it dominates every other[4]. This makes Levin's universal distribution the optimal prior distribution when no other information about the data is available, and the ultimate optimal predictor (Solomonoff's original motivation[31] was actually to capture the notion of learning by inference) when assuming the process to be algorithmic (or more precisely, carried out by a universal Turing machine). Hence the adjective "universal."

The algorithmic probability of a string is uncomputable. One way to calculate the algorithmic probability of a string is to calculate the universal distribution by running a large set of abstract machines producing an output distribution, as we did in [9].

8.6 The study of the real time series v. the simulation of an algorithmic market

The aim of this work is to study of the direction and eventually the magnitude to time series of real financial markets. To that mean, we first develop a codification procedure translating financial series into binary digit sequences. Despite the convenience and simplicity of the procedure, the translation captures several important features of the actual behavior of prices in financial markets. At the right level, a simplification of finite data into a binary language is always possible. Each observation measuring one or more parameters (e.g. price, trade name, etc.) is an enumeration of independent

3. A universal Turing machine is an abstraction of a general-purpose computer. Essentially, as proven by Alan Turing, a universal computer can simulate any other computer on an arbitrary input by reading both the description of the computer to be simulated and the input thereof from its own tape.

4. Since it is based on the Turing machine model, from which the adjective *universal* derives, the claim depends on the Church-Turing thesis.

distinguishable values, a sequence of discrete values translatable into binary terms [5].

8.6.1 From AIT back to the behavior of financial markets

Different market theorists will have different ideas about the likely pattern of 0's and 1's that can be expected from a sequence of price movements. Random walk believers would favor random-looking sequences in principle. Other analysts may be more inclined to believe that patterned-looking sequences can be spotted in the market, and may attempt to describe and exploit these patterns, eventually deleting them.

In an early anticipation of an application of AIT to the financial market [25], it was reported that the information content of price movements and magnitudes seem to drastically vary when measured right before crashes compared to periods where no financial turbulence is observed. As described in [25], this means that sequences corresponding to critical periods show a qualitative difference compared to the sequences corresponding to periods of stability (hence prone to be modeled by the traditional stochastic models) when the information content of the market is very low (and when looks random as carrying no information). In [25], the concept of conditional algorithmic complexity is used to measure these differences in the time series of price movements in two different financial markets (NASDAQ and the Mexican IPC), here we use a different algorithmic tool, that is the concept of algorithmic probability.

We will analyze the complexity of a sequence s of encoded price movements, as described in section 8.7.2. We will see whether this distribution approaches one produced artificially—by means of algorithmic processes—in order to conjecture the algorithmic forces at play in the market, rather than simply assume a pervasive randomness. Exploitable or not, we think that price movements may have an algorithmic component, even if some of this complexity is disguised behind apparent randomness.

According to Levin's distribution, in a world of computable processes, patterns which result from simple processes are relatively likely, while patterns that can only be produced by very complex processes are relatively

5. Seeing it as a binary sequence may seem an oversimplification of the concept of a natural process and its outcome, but the performance of a physical experiment always yields data written as a sequence of individual observations sampling certain phenomena.

unlikely. Algorithmic probability would predict, for example, that consecutive runs of the same magnitude, i.e. runs of pronounced falls and rises, and runs of alternative regular magnitudes have greater probability than random-looking changes. If one fails to discern the same simplicity in the market as is to be observed in certain other real world data sources[34], it is likely due to the dynamic of the stock market, where the exploitation of any regularity to make a profit results in the deletion of that regularity. Yet these regularities may drive the market and may be detected upon closer examination. For example, according to the classical theory, based on the average movement on a random walk, the probability of strong crashes is nil or very low. Yet in actuality they occur in cycles over and over.

What is different in economics is the nature of the dynamics some of the data is subject to, as discussed in section 8.4, which underscores the fact that patterns are quickly erased by economic activity itself, in the search for an economic equilibrium.

Assuming an algorithmic hypothesis, that is that there is a rule-based—as opposed to a purely stochastic—component in the market, one could apply the tools of the theory of algorithmic information, just as assuming random distributions led to the application of the traditional machinery of probability theory.

If this algorithmic hypothesis is true, the theory says that Levin's distribution is the optimal predictor. In other words, one could run a large number of machines to simulate the market, and m, the algorithmic probability based on Levin's universal distribution would provide accurate insights into the particular direction and magnitude of a price based on the fact that the market has a rule-based element. The correlation found in the experiments described in the next section 8.7.2 suggests that Levin's distribution may turn out to be a way to calculate and approximate this potentially algorithmic component in the market.

8.6.2 Unveiling the machinery

When observing a certain phenomenon, its outcome f can be seen as the result of a process P. One can then ask what the probability distribution of P generating f looks like. A probability distribution of a process is a description of the relative number of times each possible outcome occurs in a number of trials.

In a world of computable processes, Levin's semi-measure (a.k.a universal distribution) establishes that patterns which result from simple processes

It is not only in times of great volatility that one can see that markets are correlated to each other. This correlation means that, as may be expected, markets systematically react to each other. One can determine the information assimilation process time by looking at the correlations of sequences of daily closing prices of different lengths for five of the largest European and U.S. stock markets. It is evident that they react neither immediately nor after an interval of several days. As suggested by the table 8.7.1, over a period of 20 years, from January 1990 to January 2010, the average assimilation time is about a week to a week and a half. For one thing, the level of confidence of the correlation confirms that even if some events may be seen as randomly produced, the reactions of the markets follow each other and hence are neither independent of each other nor completely random. The correlation matrix 8.7.1 exhibits the Spearman rank correlation coefficients, followed by the number of elements compared (number of sequence lengths found in one or another market), underlining the significance of the correlation between them.

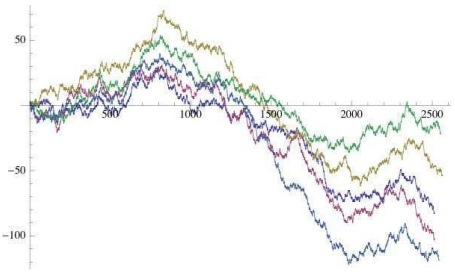

Figure 8.7: Series of daily closing prices of five of the largest stock markets from 01/01/2000 to January 01/01/2010. The best sequence length correlation suggests that markets catch up with each other (assimilate each others' information) in about 7 to 10 days on average.

(short programs) are likely, while patterns produced by complicated processes (long programs) are relatively unlikely. Unlike other probability measures, Levin's semi-measure (denoted by m) is not only a probability distribution establishing that there are some objects that have a certain probability of occurring according to said distribution, it is also a distribution specifying the order of the particular elements in terms of their individual information content.

Figure 7 suggests that by looking at the behavior of one market, the behavior of the others may be predicted. But this cannot normally be managed quickly enough for the information to be of any actual use (in fact the very intention of succeeding in one market by using information from another may be the cause rather than the consequence of the correlation).

In the context of economics, if we accept the algorithmic hypothesis (that

price changes are algorithmic, not random), m would provide the algorithmic probability of a certain price change happening, given the history of the price. An anticipation of the use of this prior distribution as an inductive theory in economics is to be found in [32]. But following that model would require us to calculate the prior distribution m, which we know is uncomputable. We proceed by approaching m experimentally in order to show what the distribution of an algorithmic market would look like, and eventually use it in an inductive framework.

Once m is approximated, it can be compared to the distribution of the outcome in the real world (i.e. the empirical data on stock market price movements). If the output of a process approaches a certain probability distribution, one accepts, within a reasonable degree of statistical certainty, that the generating process is of the nature suggested by the distribution. If it is observed that an outcome s occurs with probability $m(s)$, and the data distribution approaches m, one would be persuaded, within the same degree of certainty, to accept a uniform distribution as the footprint of a random process (where events are independent of each other), i.e. that the source of the data is suggested by the distribution m.

What Levin's distribution implies is that most rules are simple because they produce simple patterned strings, which algorithmically complex rules are unlikely to do. A simple rule, in terms of algorithmic probability, is the average kind of rule producing a highly frequent string, which according to algorithmic probability has a low random complexity (or high organised complexity), and therefore looks patterned. This is the opposite of what a complex rule would be, when defined in the same terms—it produces a pattern-less output, and is hence random-looking.

The outcomes of simple rules have short descriptions because they are less algorithmically complex, means that in some sense *simple* and *short* are connected, yet large rules may also be simple despite not being short in relative terms. And the outcomes of the application of simple rules tend to accumulate exponentially faster than the outcomes of complicated rules. This causes some events to occur more often than others and therefore to be dependent on each other. Those events happening more frequently will tend to drastically outperform other events and will do so by quickly beginning to follow the irregular pattern and doing so closely for a while.

The first task is therefore to produce a distribution by purely algorithmic means using abstract computing machines[6]—by running abstract computa-

6. One would actually need to think of one-way non-erasing Turing machines to produce a suitable distribution analogous to what could be expected from a sequence of events that

tional devices like Turing machines and cellular automata.

8.6.3 Binary encoding of the market direction of prices

Market variables have three main indicators. The first is whether there is a significant price change (i.e. larger than, say, the random walk expectation), the second is the direction of a price change (rising or falling), and thirdly, there is its magnitude. We will focus our first attempt on the direction of price changes, since this may be the most valuable of the three (after all whether one is likely to make or lose money is the first concern, before one ponders the magnitude of the possible gain or loss).

In order to verify that the market carries the algorithmic signal, the information content of the non-binary tuples can be collapsed to binary tuples. One can encode the price change in a single bit by "normalizing" the string values, with the values of the entries themselves losing direction and magnitude but capturing price changes.

Prices are subject to such strong forces (interests) in the market that it would be naive to think that they could remain the same to any decimal fraction of precision, even if they were meant to remain the same for a period of time. In order to spot the algorithmic behavior one has to provide some stability to the data by getting rid of precisely the kind of minor fluctuations that a random walk may predict. If not, one notices strong biases disguising the real patterns. For example, periods of price fluctuations would appear less likely than they are in reality if one allows decimal fluctuations to count as much as any other fluctuation [7].

This is because from one day to another the odds that prices will remain exactly the same up to the highest precision is extremely unlikely, due to the extreme forces and time exposure they are subject to. Even though it may seem that at such a level of precision one or more decimal digits could represent a real price change, for most practical purposes there is no qualitative change when prices close to the hundreds, if not actually in the hundreds, are involved. Rounding the price changes to a decimal fraction provides some stability, e.g. by improving the place of the 0^n tuple towards the top, because as we will see, it turns out to be quite well placed at the top once the

have an unrewritable past and a unique direction (the future), but the final result shouldn't be that different according to the theory.

7. The same practise is common in time-series decomposition analysis, where the sole focus of interest is the average movement, in order that trends, cycles or other potential regularities may be discerned.

decimal fluctuations have been gotten rid of. One can actually think of this as using the Brownian motion expectation to get rid of meaningless changes. In other words, the Brownian movement is stationary in our analysis, being the predictable (easy) component of the minor fluctuations, while the actual objects of study are the wilder dynamics of larger fluctuations. We have changed the focus from what analysts are used to considering as the signal, to noise, and recast what they consider noise as the algorithmic footprint in empirical data.

Detecting rising v. falling

As might be expected given the apparent randomness of the market and the dynamics to which it is subject, it would be quite difficult to discern clear patterns of rises v. falls in market prices.

The asymmetry in the rising and falling ratio is explained by the pattern deletion dynamic. While slight rises and falls have the same likelihood of occurring, making us think they follow a kind of random walk bounded by the average movement of Brownian motion, significant rises are quickly deleted by people taking advantage of them, while strong drops quickly worsen because of people trying to sell rather than buy at a bargain. This being so, one expects to see longer sequences of drops than of rises, but actually one sees the contrary, which suggests that periods of optimism are actually longer than periods of pessimism, though periods of pessimism are stronger in terms of price variation. In [25] it was reported that the information content of price movements and magnitudes seem to drastically vary when measured to intervals of high volatility (particularly right before crashes) compared to periods where no financial turbulence is observed.

We construct a binary series associated to each real time series of financial markets as follows. Let $\{p_t\}$ be the original time series of daily closing prices of a financial market for a period of time t. Then each element $b_i \in \{b_n\}$, the time series of price differences, with $n = t - 1$, is calculated as follows:

$$b_i = \begin{cases} 1 & p_{i+1} > p_i \\ 0 & p_{i+1} \leq p_i \end{cases}$$

The frequency of binary tuples of short lengths will be compared to the frequency of binary tuples of length the same length obtained by running abstract machines (deterministic Turing machines and one-dimensional cellular automata).

8.6.4 Calculating the algorithmic time series

Constructing Levin's distribution m from abstract machines is therefore necessary in order to strengthen the algorithmic hypothesis. In order to make a meaningful comparison with what can be observed in a purely rule-governed market, we will construct from the ground up an experimental distribution by running algorithmic machines (Turing machines and cellular automata). An abstract machine consists of a definition in terms of input, output, and the set of allowable operations used to turn the input into the output. They are of course algorithmic by nature (or by definition).

The Turing machine model represents the basic framework underlying many concepts in computer science, including the definition of algorithmic complexity. In this paper we use the output frequency distribution produced by running a finite, yet very large set of Turing machines with empty input as a means to approximate m. There are 11 019 960 576 four-state two-symbol Turing machines (for which the halting condition is known, thanks to the Busy Beaver game). The conception of the experiment and further details are provided in [9] and [34]. These Turing machines produced an output, from which a frequency distribution of tuples was calculated and compared to the actual time series from the market data produced by the stock markets encoded as described in 8.6.3. A forthcoming paper provides more technical details[10].

Also, a frequency distribution from a sample of 10 000 four-color totalistic cellular automata was built from a total of 1 048 575 possible four-color totalistic cellular automata, each rule starting from an random initial configuration of 10 to 20 black and white cells, and running for 100 steps (hence arbitrarily halted). Four-color totalistic cellular automata produce four-symbol sequences. However only the binary were taken into consideration, with the purpose of building a binary frequency distribution. The choice of this cellular automata (CA) space was dictated by the fact that the smallest CA space is too small, and the next smallest space too large to extract a significant enough sample from it. Having chosen a sample of four-color totalistic CA, the particular rules sample was randomly generated.

For all the experiments, stock market datasets (of daily closing prices) used covered the same period of time: from January 1990 through January 2010. The stock market code names can be readily connected with their full index names or symbols.

Spearman's rank coefficient was the statistical measure of the correlation between rankings of elements in the frequency distributions. As is known,

if there are no repeated data values, a perfect Spearman correlation of $+1$ or -1 occurs when each of the variables is a perfect monotone function of the other. While 1 indicates perfect correlation (i.e. in the exact order), -1 indicates perfect negative correlation (i.e. perfect inverse order).

8.7 Experiments and Results

It is known that for any finite series of a sequence of integer values, there is a family of countable infinite computable functions that fit the sequence over its length. Most of them will be terribly complicated, and any attempt to find the simplest accounting for the sequence will face uncomputability. Yet using the tools of algorithmic information theory, one can find the simplest function for short sequences by deterministic means, testing a set of increasingly complex functions starting from the simplest and proceeding until the desired sequence assuring the simplest fit is arrived at. Even if the likelihood of remaining close to the continuation of the series remains low, one can recompute the sequence and find good approximations for short periods of time.

The following experiment uses the concept of algorithmic complexity to find the best fit in terms of the simplest model fitting the data, assuming the source to be algorithmic (rule-based).

8.7.1 Rankings of price variations

In the attempt to capture the price change phenomenon in the stock market, we have encoded price changes in binary strings. The following are tables capturing the rise and fall of prices and their occurrence in a ranking classification.

Carrying an algorithmic signal

Some regularities and symmetries in the sequence distribution of price directions in the market may be accounted for by an algorithmic signal, but they also follow from a normal distribution. For example, the symmetry between the left and right sides of the Gaussian curve with zero skewness means that reverted and inverted strings (i.e. consecutive runs of events of prices going up or down) will show similar frequency values, which is also in keeping with the intuitive expectation of the preservation of complexity

invariant to certain basic transformations (a sequence of events 1111... are equally likely to occur as 0000..., or 0101... and 1010...).

This means that one would expect n consecutive runs of rising prices to be as likely as n consecutive runs of falling prices. Some symmetries, however, are broken in particular scenarios. In the stock market for example, it is very well known that sequences of drastic falls are common from time to time, but never sequences of drastic price increases, certainly not increases of the same magnitude as the worst price drops. And we witnessed such a phenomenon in the distributions from the DJI. Some other symmetries may be accounted for by business cycles engaged in the search for economic equilibrium.

Correlation matrix: market v. market

market v. market	4	5	6	7	8	9	10
CAC40 v. DAX	0.059|16	0.18|32	0.070|62	0.37|109	0.48|119	0.62|87	0.73|55
CAC40 v. DJIA	0.31|16	0.25|32	0.014|62	0.59|109	0.27|124	0.34|95	0.82|51
CAC40 v. FTSE350	0.16|16	−0.019|32	−0.18|63	0.15|108	0.59|114	0.72|94	0.73|62
CAC40 v. NASDAQ	0.30|16	0.43|32	0.056|63	0.16|111	0.36|119	0.32|88	0.69|49
CAC40 v. SP500	0.14|16	0.45|31	−0.085|56	−0.18|91	0.16|96	0.49|73	0.84|45
DAX v. DJIA	0.10|16	−0.14|32	0.13|62	0.37|110	0.56|129	0.84|86	0.82|58
DAX v. FTSE350	0.12|16	−0.029|32	0.12|63	0.0016|106	0.54|118	0.81|89	0.80|56
DAX v. NASDAQ	0.36|16	0.35|32	0.080|62	0.014|110	0.64|126	0.55|96	0.98|48
DAX v. SP500	0.38|16	0.062|31	−0.20|56	−0.11|88	0.11|94	0.43|76	0.63|49
DJIA v. FTSE350	0.35|16	−0.13|32	−0.022|63	0.29|107	0.57|129	0.76|99	0.86|56
DJIA v. NASDAQ	−0.17|16	−0.13|32	0.0077|62	0.079|112	0.70|129	0.57|111	0.69|64
DJIA v. SP500	−0.038|16	0.32|31	−0.052|55	0.14|89	0.37|103	0.32|86	0.60|59
FTSE350 v. NASDAQ	0.36|16	0.38|32	−0.041|63	0.54|108	0.68|126	0.57|107	0.66|51
FTSE350 v. SP500	0.50|16	0.50|31	0.12|56	−0.11|92	0.25|101	0.26|96	0.29|66
NASDAQ v. SP500	0.70|16	0.42|31	0.20|56	0.024|91	0.41|111	0.23|102	0.42|61

Table 1. Stock market v. stock market.

With algorithmic probability in hand, one may predict that alternations and consecutive events of the same type and magnitude are more likely, because they may be algorithmically more simple. One may, for example, expect to see symmetrical events occurring more often, with reversion and complementation occurring together in groups (i.e. a string 10^n occurring together with 0^n1 and the like). In the long-term, business cycles and economic equilibria may also be explained in information theoretic terms, because for each run of events there are the two complexity-preserving symmetries, reversion and complementation, that always follow their counterpart sequences (the unreversed and complement of the complement), producing a cyclic type of behavior.

The correlations shown in table 8.7.1 indicate what is already assumed in looking for cycles and trends, viz. that these underlying cycles and trends in

the markets are more prominent when *deleting* Brownian *noise*. As shown below, this may be an indication that the tail of the distribution has a stronger correlation than the elements covered by the normal curve, as could be inferred from the definition of a random walk (that is, that random walks are not correlated at all).

The entries in each comparison table consist of the Spearman coefficient followed by the number of elements compared. Both determine the level of confidence and are therefore essential for estimating the correlation. Rows compare different stock markets over different sequence lengths of daily closing prices, represented by the columns. It is also worth noting when the comparison tables, such as 8.7.1, have no negative correlations.

Rounded market v. rounded market

market v. market	4	5	6	7	8	9	10
CAC40 v. DAX	0.58|11	0.58|15	0.55|19	0.37|26	0.59|29	0.55|31	0.60|28
CAC40 v. DJIA	0.82|11	0.28|15	0.28|21	0.29|24	0.52|29	0.51|32	0.33|36
CAC40 v. FTSE350	0.89|11	0.089|15	0.41|20	0.17|22	0.59|30	0.28|28	0.30|34
CAC40 v. NASDAQ	0.69|11	0.27|14	0.28|18	0.44|23	0.30|30	0.17|34	0.61|30
CAC40 v. SP500	0.85|11	0.32|15	0.49|20	0.55|24	0.42|33	0.35|35	0.34|36
DAX v. DJIA	0.76|11	0.45|16	0.56|20	0.35|26	0.34|28	0.25|35	0.24|33
DAX v. FTSE350	0.61|11	0.30|16	0.58|19	0.14|25	0.30|29	0.34|30	0.21|31
DAX v. NASDAQ	0.40|11	0.27|16	0.36|18	0.75|25	0.28|29	0.28|35	0.50|33
DAX v. SP500	0.14|12	0.36|17	0.72|20	0.64|28	0.42|31	0.52|34	0.51|32
DJIA v. FTSE350	0.71|11	0.30|16	0.63|20	0.71|22	0.21|28	0.28|31	0.35|33
DJIA v. NASDAQ	0.58|11	0.52|15	0.33|19	0.58|23	0.46|29	0.49|37	0.51|35
DJIA v. SP500	0.70|11	0.20|16	0.45|21	0.29|26	0.35|32	0.37|36	0.55|36
FTSE350 v. NASDAQ	0.73|11	0.57|15	0.70|17	0.48|23	0.62|28	0.34|33	0.075|35
FTSE350 v. SP500	0.66|11	0.65|16	0.56|19	0.18|24	0.64|32	0.32|32	0.52|38
NASDAQ v. SP500	0.57|11	0.37|15	0.41|18	0.32|24	0.30|34	0.19|35	0.35|40

Table 2. In contrast, when the markets are compared to random price movements, which accumulate in a normal curve, they exhibit no correlation or only a very weak correlation, as shown in 8.7.1:

Market v. random

r. market v. random	4	5	6	7	8	9	10
DJIA v. random	−0.050|16	0.080|31	−0.078|61	0.065|96	0.34|130	0.18|120	0.53|85
SP500 v. random	0.21|16	−0.066|30	0.045|54	−0.16|81	0.10|99	0.29|87	0.32|57
NASDAQ v. random	0.12|16	−0.095|31	0.11|60	0.14|99	0.041|122	0.29|106	0.57|68
FTSE350 v. random	0.16|16	−0.052|31	0.15|61	0.14|95	0.30|122	0.50|111	0.37|77
CAC40 v. random	0.32|16	−0.15|31	−0.13|60	0.16|99	0.19|119	0.45|109	0.36|78
DAX v. random	0.33|16	0.023|31	0.20|60	0.14|95	0.26|129	0.31|104	0.31|77

Table 3. When random price movements are compared to rounded prices of the market (denoted by "r. market" to avoid accumulation in the normal

curve) the correlation coefficient is too weak, possessing no significance at all. This may indicate that it is the prices behaving and accumulating in the normal curve that effectively lead the overall correlation.

Rounded market v. random

market v. random	4	5	6	7	8	9	10
DJIA v. random	0.21|12	−0.15|17	0.19|23	−0.033|28	−0.066|33	0.31|29	0.64|15
SP500 v. random	−0.47|12	−0.098|17	−0.20|25	0.32|31	0.20|38	0.41|29	0.32|20
NASDAQ v. random	−0.55|11	−0.13|16	−0.093|20	0.18|26	0.015|37	0.30|35	0.38|25
FTSE350 v. random	−0.25|11	−0.24|16	−0.053|22	−0.050|24	−0.11|31	0.25|23	0.49|13
CAC40 v. random	−0.12|11	−0.14|15	0.095|22	0.23|26	0.18|30	0.36|23	0.44|14
DAX v. random	0.15|12	−0.067|18	−0.12|24	0.029|31	0.31|32	0.27|27	0.59|15

Table 4. Comparison between the daily stock market sequences v. an hypothesised log-normal accumulation of price directions.

Market v. Turing machines

market v. TM	5	6	7	8	9	10
DJIA v. TM	0.42|16	0.20|21	0.42|24	−0.021|35	−0.072|36	0.20|47
SP500 v. TM	0.48|18	0.30|24	−0.070|32	0.32|39	0.26|47	0.40|55
NASDAQ v. TM	0.67|17	0.058|25	0.021|32	0.26|42	0.076|49	0.17|57
FTSE350 v. TM	0.30|17	0.39|22	0.14|29	0.43|36	0.013|41	0.038|55
CAC40 v. TM	0.49|17	0.026|25	0.41|32	0.0056|38	0.22|47	0.082|56

Table 5. The comparison to TM revealed day lengths better correlated than other, although
their significance remained weak and unstable, with a tendency, however, to positive correlations.

Market v. cellular automata

market v. CA	4	5	6	7	8	9	10
DJIA v. CA	−0.14|16	0.28|32	−0.084|63	−0.049|116	0.10|148	0.35|111	0.51|59
SP500 v. CA	−0.16|16	0.094|32	0.0081|64	0.11|116	0.088|140	0.17|117	0.40|64
NASDAQ v. CA	0.065|16	0.25|32	0.19|63	0.098|116	0.095|148	0.065|131	0.36|65
FTSE350 v. CA	−0.16|16	−0.15|32	0.12|64	−0.013|120	−0.0028|146	0.049|124	0.42|76
CAC40 v. CA	−0.035|16	0.36|32	0.21|64	0.064|114	0.20|138	0.25|114	0.33|70

Table 6. When compared to the distribution from cellular automata, the correlation was greater. Each column had pairs of score means: (-0.09, 16), (0.17, 32), (0.09, 64), (0.042, 116), (0.096, 144), (0.18, 119), (0.41, 67) for 4 to 10 days, for which the last 2 (9 and 10 days long) have significant levels of correlation according to their critical values and the number of elements compared.

8.7.2 Backtesting

Applying the same methodology over a period of a decade, from 1980 to 1990 (old market), to three major stock markets for which we had data for the said period of time, similar correlations were found across the board, from weak to moderately weak— though the trend was always toward positive correlations.

Old market v. CA distribution

old market v. CA	4	5	6	7	8	9	10
DJIA v. CA	0.33|10	0.068|16	0.51|21	0.15|28	−0.13|31	0.12|32	0.25|29
SP500 v. CA	0.044|13	0.35|19	0.028|24	0.33|33	0.45|33	0.00022|30	0.37|34
NASDAQ v. CA	0.45|10	0.20|17	0.27|24	0.16|30	0.057|31	0.11|34	0.087|32

Table 7. Comparison matrix of frequency distributions of daily price directions of three stock markets from 1980 to 1990.

The distributions indicate that price changes are unlikely to rise by more than a few points for more than a few days, while greater losses usually occur together and over longer periods. The most common sequences of changes are alternations. It is worth noticing that sequences are grouped together by reversion and complementation relative to their frequency, whereas traditional probability would have them occur in no particular order and with roughly the same frequency values.

Tables 8 and 9 illustrate the kind of frequency distributions from the stock markets (in this case for the DJI) over tuples of length 3 with which distributions from the market data were compared with and its statistical correlation evaluated section between four other stock markets and over larger periods of time up to 10 closing daily prices.

tuple	*prob.*
000	0.139
001	0.130
111	0.129
011	0.129
100	0.129
110	0.123
101	0.110
010	0.110

Table 8. 3-tuples distribution from the DJI price difference time series for the past 80 years. 1 means that a price rose, 0 that it fell or remained the

same as described in the construction of the binary sequence b_n as described in 8.6.3 and partitioned in 3-tuples for this example. By rounding to the nearest multiple of .4 (i.e. dismissing decimal fraction price variations of this order) some more stable patterns start to emerge.

tuple	*prob.*
000	0.00508
111	0.00508
001	0.00488
011	0.00488
100	0.00488
110	0.00468
010	0.00483
101	0.00463

Table 9. 3-tuples from the output distribution produced by running all 4-state
2-symbol Turing machines starting from an empty tape first on a background of 0's and then running it again on a background of 1s to avoid asymmetries due to the machine formalism convention.

8.7.3 Algorithmic inference of rounded price directions

Once with the tuples distributions calculated and a correlation found, one can apply Solomonoff's[31] concept algorithmic inference. Let's say that by looking two days behind of daily closing prices one sees two consecutive losses. The algorithmic inference will say that with probability 0.129 the third day will be a loss again. In fact, as we now know, algorithmic probability will suggest that with higher probability the next day will only repeat the last values of any run of 1's or 0's and the empirical distribution from the market will tell us that runs of 1s (gains) are more likely than consecutive losses (before the rounding process deleting the smallest price movements) without taking into account their magnitude (as empirically known, losses are greater than gains, but gains are more sustainable), but runs of consecutive 0's (losses) will be close or even more likely than consecutive losses after the rounding process precisely because gains are smaller in magnitude.

To calculate the algorithmic probability of a price direction b_i of the next closing price by looking n consecutive daily rounded prices behind, is given by:

$$P(b_i) = m(b_{i-n} \ldots b_i)$$

i.e. the algorithmic probability of the string constituted by the n consecutive price directions of the days before followed by the possible outcome to be estimated, with m Levin's semi-measure described in equation 8.7.3. It is worth notice however that the inference power of this approach is limited by the correlation found between the market's long tails and the distributions calculated by means of exhaustive computation. Tables with distributions for several tuple lengths and probability values will be available in [10], so that one can empirically calculate m by means of the results of exhaustive computation.

For example, the algorithmic model predicts a greater incidence of simple signatures as trends under the *noise* modeled by the Brownian motion model, such as signatures 000... of price stability. It also predicts that random-looking signatures of higher volatility will occur more if they are already occurring, a signature in unstable times where the Brownian motion no longer works in these kind of events outside the main Bell curve.

8.8 Further considerations

8.8.1 Rule-based agents

For sound reasons, economists are used to standardizing their discussions by starting out from certain basic assumptions. One common assumption in economics is that actors in the market are decision makers, often referred to as *rational agents*. According to this view, rational agents make choices by assessing possible outcomes and assigning a utility to each in order to make a decision. In this rational choice model, all decisions are arrived at by a *rational* process of weighing costs against benefits, and not randomly.

An agent in economics or a player in game theory is an actor capable of decision making. The idea is that the agent initiates actions, given the available information, and tries to maximize his or her chances of success (traditionally their personal or collective utilities), whatever the ultimate goal may be. The algorithm that each agent takes may be non-deterministic, which means that the agent may make decisions based on probabilities, not that at any stage of the process a necessarily truly random choice is made. It actually doesn't matter much whether their actions may be perceived as mistaken, or their utility questioned. What is important is that agents follow

rules, or if any chance is involved there is another large part in it not random at all (specially when one takes into consideration the way algorithmic trading is done). The operative assumption is that the individual has the cognitive ability to weigh every choice he/she makes, as opposed to taking decisions stochastically. This is particularly true when there is nothing else but computers making the decisions.

This view, wherein each actor can be viewed as a kind of automaton following his or her own particular rules, does not run counter to the stance we adopt here, and it is in perfect agreement with the algorithmic approach presented herein (and one can expect the market to get more algorithmic as more automatization is involved). On the contrary, what we claim is that if this assumption is made, then the machinery of the theory of computation can be applied, particularly the theory of algorithmic information (AIT). Hence market data can be said to fall within the scope of algorithmic probability.

Our approach is also compatible with the emergent field of behavioral economics[4], provided the set of cognitive biases remain grounded in rules. Rules followed by emotional (or non-rational) traders can be as simple as imitating behavior, repeating from past experience, acting out of fear, taking advice from others or following certain strategy. All these are algorithmic in nature in that they are rule based, despite their apparent idiosyncrasy (assuming there are no real clairvoyants with true metaphysical powers). Even though they may look random, what we claim, on the basis of algorithmic probability and Levin's distribution, is that most of these behaviors follow simple rules. It is the accumulation of simple rules rather than the exceptional complicated ones which actually generate trends.

If the market turns out to be based on simple rules and driven by its intrinsic complexity rather than by the action of truly random external events, the choice or application of rational theory would be quite irrelevant. In either case, our approach remains consistent and relevant. Both the rational and, to a large extent, the behavioral agent assumptions imply that what we are proposing here is that algorithmic complexity can be directly applied to the field of market behavior, and that our model comes armed with a natural toolkit for analyzing the market, viz. algorithmic probability.

8.8.2 The problem of over-fitting

When looking at a set of data following a distribution, one can claim, in statistical terms, that the source generating the data is of the nature that the distribution suggests. Such is the case when a set of data follows a

model, where depending on certain variables, one can say with some degree of certitude that the process generating the data follows the model.

It seems to be well-known and largely accepted among economists that one can basically fit anything to anything else, and that this has shaped most of the research in the field, producing a sophisticated toolkit dictating how to achieve this fit as well as how much of a fit is necessary for particular purposes, even though such a fit may have no relevance either to the data or to particular forecasting needs, being merely designed to produce an instrument with limited scope fulfilling a specific purpose.

However, a common problem is the problem of over-fitting, that is, a false model that may fit perfectly with an observed phenomenon. A statistical comparison cannot actually be used to categorically prove or disprove a difference or similarity, only to favour one hypothesis over another.

To mention one of the arbitrary parameters that we might have taken, there is the chosen rounding. We found it interesting that the distributions from the stock markets were sometimes unstable to the rounding process of prices. Rounding to the closest .4 was the threshold found to allow the distribution to stabilise. This instability may suggest that there are two different kinds of forces acting, one producing very small and likely negligible price movements (in agreement to the random walk expectation), and other producing the kind of qualitative changes in the direction of prices that we were interested in. In any case this simply results in the method only being able to predict changes of the order of magnitude of the rounding proceeding from the opposite direction, assuming that the data is not random, unlike the stochastic models.

Algorithmic probability rests upon two main principles: the principle of multiple explanations, which states that one should keep all hypotheses that are consistent with the data, and a second principle known as Occam's razor, which states that when inferring causes, entities should not be multiplied beyond necessity, or, alternatively, that among all hypotheses consistent with the observations, the simplest should be favoured. As for the choice of an a priori distribution over a hypothesis, this amounts to assigning simpler hypotheses a higher probability and more complex ones a lower probability. So this is where the concept of algorithmic complexity comes into play.

As proven by Levin and Solomonoff, the algorithmic probability measure (the universal distribution) will outperform any other, unless other information is available that helps to foresee the outcome, in which case an additional variable could be added to the model to account for this information. But since we've been suggesting that information will propagate fast enough

even though the market is not stochastic in nature, deleting the patterns and making them unpredictable, any additional assumption only complicates the model. In other words, Levin's universal distribution is optimal over all non-random distributions[21], in the sense that the algorithmic model is by itself the simplest model fitting the data when this data is produced by a process (as opposed to being randomly generated). The model is itself ill-suited to an excess of parameters argument because it basically assumes only that the market is governed by rules.

As proven by Solomonoff and Levin, any other model will simply overlook some of the terms of the algorithmic probability sum. So rather than being more precise, any other model will differ from algorithmic probability in that it will necessarily end up overlooking part of the data. In other words, there is no better model taking into account the data than algorithmic probability. As Solomonoff has claimed, one can't do any better. Algorithmic inference is a time-limited optimisation problem, and algorithmic probability accounts for it simply.

8.9 Conclusions and further work

When looking at a large-enough set of data following a distribution, one can in statistical terms safely assume that the source generating the data is of the nature that the distribution suggests. Such is the case when a set of data follows a normal distribution, where depending on certain statistical variables, one can, for example, say with a high degree of certitude that the process generating the data is of a random nature. If there is an algorithmic component in the empirical data of price movements in financial markets, as might be suggested by the distribution of price movements, algorithmic information theory may account for the deviation from log-normality as argued herein. In the words of Velupillai[32]—quoting Clower[6] talking about Putnam's approach to a theory of induction[28][27]—*This may help ground "economics as an inductive science"* again.

One may well ask whether a theory which assumes that price movements follow an algorithmic trend ought not to be tested in the field to see whether it outperforms the current model. The truth is that the algorithmic hypothesis would easily outperform the current model, because it would account for recurrent periods of instability. The current theory, with its emphasis on short term profits, is inclined to overlook these, for reasons that are probably outside the scope of scientific inquiry. In our understanding, the profits attributed to the standard current model are not really owed to the model as

such, but rather to the mechanisms devised to control the risk-taking inspired by the overconfidence that the model generates.

In a strict sense, this paper describes the ultimate possible numerical simulation of the market when no further information about it is known (or cannot be known in practise) assuming no other (neither efficient markets nor general equilibrium) but actors following a set of rules and therefore to behave algorithmically at least at some extent hence potentially modeled by algorithmic probability.

The simulation may turn out to be of limited predictive value—for looking no more than a few days ahead and modeling weak signals—due to the deleting patterns phenomenon (i.e. the time during which the market assimilates new information). More experiments remain to be done which carefully encode and take into consideration other variables, such as the magnitude of prices, for example, looking at consecutive runs of gains or loses.

Acknowledgments

Hector Zenil wishes to thank Vela Velupillai and Stefano Zambelli for their kind invitation to take part in the workshop on Nonlinearity, Complexity and Randomness at the Economics Department of the University of Trento, and for their useful comments. Jason Cawley, Fred Meinberg, Bernard François and Raymond Aschheim provided helpful suggestions, for which many thanks. And to Ricardo Mansilla for pointing us out to his own work and kindly provided some datasets. Any misconceptions remain of course the sole responsibility of the authors.

Bibliography

[1] Arora, S., Barak, B., Brunnermeier, M. and Rong, G. (2009) *Computational Complexity and Information Asymmetry in Financial Products.* working paper.

[2] Bachelier, L. (1900) *Théorie de la spéculation.* Annales Scientifiques de l'Ecole Normale Supérieure, 3 (17): 21-86.

[3] Calude, C.S. (2002) *Information and Randomness: An Algorithmic Perspective (Texts in Theoretical Computer Science. An EATCS Series).* Springer, 2nd. edition.

[4] Camerer, C., Loewenstein, G. and Rabin, M. *Advances in Behavioral Economics,* Princeton University Press.

[5] Chaitin, G.J. (2001) *Exploring Randomness,* Springer Verlag.

[6] Clower, R. W. (1994) *Economics as an inductive science,* Southern Economic Journal, 60: 805-14.

[7] Cont, R. and Tankov, P. (2003) *Financial modeling with Jump Processes,* Chapman & Hall/CRC Press.

[8] Downey, R.G. and Hirschfeldt, D. 2010 *Algorithmic Randomness and Complexity.* Springer Verlag, 2010.

[9] Delahaye, J.P. and Zenil, H. (2007) On the Kolmogorov-Chaitin complexity for short sequences, in Calude, C.S. (eds) *Complexity and Randomness: From Leibniz to Chaitin.* World Scientific.

[10] Delahaye, J.P. and Zenil, H. *Numerical Evaluation of Algorithmic Complexity for Short Strings: A Glance into the Innermost Structure of Randomness*, arXiv:1101.4795v1, 2011.

[11] Fama, E. (1963) *Mandelbrot and the Stable Paretian Hypothesis.* The Journal of Business, Volume 36, Issue 4, 420-42.

[12] Fama, E. (1965) *The Behavior of Stock Market Prices.* Journal of Business 38: 34-105.

[13] Good, P.I. (2005) *Permutation, Parametric and Bootstrap Tests of Hypotheses*, 3rd ed., Springer.

[14] Hutter, M. (2007) *On Universal Prediction and Bayesian Confirmation.* Theoretical Computer Science, 384:1 33-48.

[15] Jaynes, E.T. (2003) *Probability Theory: The Logic of Science.* Cambridge University Press.

[16] Keynes, J.M. (1936) *General Theory of Employment Interest and Money.* London: Macmillan (reprinted 2007).

[17] Kirchherr, W. and Li, M. (1997) *The miraculous universal distribution.* Mathematical Intelligencer.

[18] Kolmogorov, A.N. (1965) *Three approaches to the quantitative definition of information.* Problems of Information and Transmission, 1(1):1—7.

[19] Lamper, D., Howison, S. and Johnson, N. F. (2002) *Predictability of large future changes in a competitive evolving population.* Phys. Rev. Lett. 88:11.

[20] Levin, L. (1977) *On a concrete method of Assigning Complexity Measures.* Doklady Akademii nauk SSSR, vol.18(3), pp. 727-731.

[21] L. Levin., (1973) Universal Search Problems., 9(3):265-266, (submitted: 1972, reported in talks: 1971). English translation in: B.A.Trakhtenbrot. *A Survey of Russian Approaches to Perebor (Brute-force Search) Algorithms.* Annals of the History of Computing 6(4):384-400.

[22] Li, M. and Vitányi, P. (1997) *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, 3rd. (revised edition, 2008).

[23] Malkiel, B. G. (2003) *A random walk down Wall Street : the time-tested strategy for successful investing,* W.W. Norton.

[24] Mandelbrot, B.B. (1963) *The variation of certain speculative prices.* The Journal of Business of the University of Chicago: 36, 394-419.

[25] Mansilla, R. (2001) *Algorithmic complexity in real financial markets.* Physica A, 301, 483-492.

[26] P. Martin-Löf, *The definition of random sequences.* Information and Control Volume 9, Issue 6: 602-619, 1966.

[27] Putnam, H. (1990) *The meaning of the concept of probability in application to finite sequences.* Garland Publishing.

[28] Putnam, H. (1975) *Probability and confirmation.* Mathematics, Matter and Methods: Philosophical papers I, Cambridge University Press.

[29] Samuelson, P. (1947) *Foundations of Economic Analysis.* Harvard University Press. footnotesize

[30] Schnorr, C.-P. (1971) *Zufälligkeit und Wahrscheinlichkeit. Eine algorithmische Begründung der Wahrscheinlichkeitstheorie.* Springer, Berlin.

[31] Solomonoff, R. J. (1964) *A formal theory of inductive inference: Parts 1 and 2,* Information and Control, 7:1—22 and 224—254.

[32] Vela Velupillai, K. (2000) *Computable Economics.* Oxford University Press.

[33] Wolfram, S. (2002) *A New Kind of Science.* Wolfram Media.

[34] Zenil H. and Delahaye, J.P. (2010) On the algorithmic nature of the world. Forthcoming in Dodig-Crnkovic, G. and Burgin, M. *Information and Computation.* World Scientific.

Chapitre 9

Program-size Versus Time Complexity : Slowdown and Speed-up Phenomena in the Micro-cosmos of Small Turing Machines

From J. Joosten, F. Soler-Toscano and H. Zenil, *Program size Versus Time Complexity: Slowdown and Speed-up Phenomena in the Micro-cosmos of Small Turing Machines*, 3rd. International Workshop on Physics and Computation 2010 Conference Proceedings, pages 175-198. Also forthcoming in the International Journal of Unconventional Computing.

9.1 Introduction

Among the several measures of computational complexity there are measures focusing on the minimal description of a program and others quantifying the resources (space, time, energy) used by a computation. This paper is a reflection of an ongoing project with the ultimate goal of contributing to the

understanding of relationships between various measures of complexity by means of computational experiments. In particular in the current paper we did the following.

We focused on small Turing Machines and looked at the kind of functions that are computable on them focussing on the runtimes. We then study how allowing more computational resources in the form of Turing machine states affect the runtimes of TMs computing these functions. We shall see that in general and on average, more resources leads to slower computations. In this introduction we shall briefly introduce the main concepts central to the paper.

9.1.1 Two measures of complexity

The long run aim of the project focuses on the relationship between various complexity measures, particularly descriptional and computational complexity measures. In this subsection we shall briefly and informally introduce them.

In the literature there are results known to theoretically link some complexity notions. For example, in [6], runtime probabilities were estimated based on Chaitin's heuristic principle as formulated in [5]. Chaitin's principle is of descriptive theoretic nature and states that *the theorems of a finitely-specified theory cannot be significantly more complex than the theory itself.*

Bennett's concept of logical depth combines the concept of time complexity and program-size complexity [1, 2] by means of the time that a decompression algorithm takes to decompress an object from its shortest description.

Recent work by Neary and Woods [16] has shown that the simulation of cyclic tag systems by cellular automata is effected with a polynomial slowdown, setting a very low threshold of possible non-polynomial tradeoffs between program-size and computational time complexity.

Computational Complexity

Computational complexity [4, 11] analyzes the difficulty of computational problems in terms of computational resources. The computational time complexity of a problem is the number of steps that it takes to solve an instance of the problem using the most efficient algorithm, as a function of the size of the representation of this instance.

As widely known, the main open problem with regard to this measure of complexity is the question of whether problems that can be solved in non-deterministic polynomial time can be solved in deterministic polynomial time, aka the P versus NP problem. Since P is a subset of NP, the question is whether NP is contained in P. If it is, the problem may be translated as, for every Turing machine computing an NP function there is (possibly) another Turing machine that does so in P time. In principle one may think that if in a space of all Turing machines with a certain fixed size there is no such a P time machine for the given function (and because a space of smaller Turing machines is always contained in the larger) only by adding more resources a more efficient algorithm, perhaps in P, might be found. We shall see that adding more resources almost certainly yields to slow-down.

Descriptional Complexity

The algorithmic or program-size complexity [10, 5] of a binary string is informally defined as the shortest program that can produce the string. There is no algorithmic way of finding the shortest algorithm that outputs a given string

More precisely, the complexity of a bit string s is the length of the string's shortest program in binary on a fixed universal Turing machine. A string is said to be complex or random if its shortest description cannot be much more shorter than the length of the string itself. And it is said to be simple if it can be highly compressed. There are several related variants of algorithmic complexity or algorithmic information.

In terms of Turing machines, if M is a Turing machine which on input i outputs string s, then the concatenated string $\langle M, i\rangle$ is a description of s. The size of a Turing machine in terms of the number of states (s) and colors (k) (aka known as symbols) can be represented by the product $s \cdot k$. Since we are fixing the number of colors to $k = 2$ in our study, we increase the number of states s as a mean for increasing the program-size (descriptional) complexity of the Turing machines in order to study any possible tradeoffs with any of the other complexity measures in question, particularly computational (time) complexity.

9.1.2 Turing machines

Throughout this project the computational model that we use will be that of Turing machines. Turing machines are well-known models for universal

computation. This means, that anything that can be computed at all, can be computed on a Turing machine.

In its simplest form, a Turing machine consists of a two-way infinite tape that is divided in adjacent cells. Each cell can be either blank or contain a non-blank color (symbol). The Turing machine comes with a "head" that can move over the cells of the tape. Moreover, the machine can be in various states. At each step in time, the machine reads what color is under the head, and then, depending on in what state it is writes a (possibly) new color in the cell under the head, goes to a (possibly) new state and have the head move either left or right. A specific Turing machine is completely determined by its behavior at these time steps. One often speaks of a transition rule, or a transition table. Figure 9.1 depicts graphically such a transition rule when we only allow for 2 colors, black and white and where there are two states, State 1 and State 2.

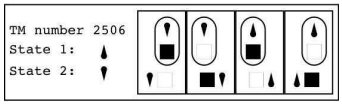

Figure 9.1: Transition table of a 2-color 2-state Turing machine with Rule 2506 according to Wolfram's enumeration and Wolfram's visual representation style [14]. [8].

For example, the head of this machine will only move to the right, write a black color and go to State 2 whenever the machine was in State 2 and it read a blank symbol.

We shall often refer to the collection of TMs with k colors and s states as a TM space. From now on, we shall write (2,2) for the space of TMs with 2 states and 2 colors, and (3,2) for the space of TMs with 3 states and 2 colors, etc.

9.1.3 Relating notions of complexity

We relate and explore throughout the experiment the connections between descriptional complexity and time computational complexity. One way to increase the descriptional complexity of a Turing machine is enlarging its transition table description by adding a new state. So what we will do is, look at time needed to perform certain computational tasks first with only 2 states, and next with 3 and 4 states.

Our current findings suggest that even if a more efficient Turing machine algorithm solving a problem instance may exist, the probability of picking a machine algorithm at random among the TMs that solve the problem in a faster time has probability close to 0 because the number of slower Turing machines computing a function outnumbers the number of possible Turing machines speeding it up by a fast growing function.

9.1.4 Investigating the micro-cosmos of small Turing machines

We know that small programs are capable of great complexity. For example, computational universality occurs in cellular automata with just 2 colors and nearest neighborhood (Rule 110, see [14, 3]) and also (weak) universality in Turing machines with only 2 states and 3 colors [15].

For all practical purposes one is restricted to perform experiments with small Turing machines (TMs) if one pursuits a thorough investigation of complete spaces for a certain size. Yet the space of these machines is rich and large enough to allow for interesting and insightful comparison, draw some preliminary conclusions and shed light on the relations between measures of complexity.

As mentioned before, in this paper, we look at TMs with 2 states and 2 colors and compare them to TMs more states. The main focus is on the functions they compute and the runtimes for these functions However, along our investigation we shall deviate from time to time from our main focus and marvel at the rich structures present in what we like to refer to as *the micro-cosmos of small Turing machines*. Like, what kind of, and how many functions are computed in each space? What kind of runtimes and space-usage do we typically see and how are they arranged over the TM space? What are the sets that are definable using small Turing machines? How many input values does one need to fully determine the function computed by a TM? We find it amazing how rich the encountered structures are even when we use so few resources.

9.1.5 Plan of the paper

After having introduced the main concepts of this paper and after having set out the context in this section, the remainder of this paper is organized as follows. In Section 9.2 we will in full detail describe the experiment,

its methodology and the choices that were made leading us to the current methodology. In Section 9.3 we present the structures that we found in (2,2). The main focus is on runtimes but a lot of other rich structures are exposed there. In Section 9.4 we do the same for the space (3,2). Section 9.5 deals with (4,2) but does not disclose any additional structure of that space as we did not exhaustively search this space. Rather we sampled from this space looking for functions we selected from (3,2). In Section 9.6 we compare the various TM spaces focussing on the runtimes of TMs that compute a particular function.

9.2 Methodology and description of the experiment

In this section we shall briefly restate the set-up of our experiment to then fill out the details and motivate our choices. We try to be as detailed as possible for a readable paper. For additional information, source code, figures and obtained data can be requested from any of the authors.

9.2.1 Methodology in short

It is not hard to see that any computation in (2,2) is also present in (3,2). At first, we look at TMs in (2,2) and compare them to TMs in (3,2). In particular we shall study the functions they compute and the time they take to compute in each space.

The way we proceeded is as follows. We ran all the TMs in (2,2) and (3,2) for 1000 steps for the first 21 input values $0, 1, \ldots, 20$. If a TM does not halt by 1000 steps we simply say that it diverges. We saw that certain TMs defined a regular progression of runtimes that needed more than 1000 steps to complete the calculation for larger input values. For these regular progressions we filled out the values manually as described in Subsection 9.2.7. Thus, we collect all the functions on the domain $[0, 20]$ computed in (2,2) and (3,2) and investigate and compare them in terms of run-time, complexity and space-usage. We selected some interesting functions from (2,2) and (3,2). For these functions we searched by sampling for TMs in (4,2) that compute them so that we could include (4,2) in our comparison.

Clearly, at the outset of this project we needed to decide on at least the following issues:

1. How to represent numbers on a TM?
2. How to decide which function is computed by a particular TM.
3. Decide when a computation is considered finished.

The next subsections will fill out the details of the technical choices made and provide motivations for these choices. Our set-up is reminiscent of and motivated by a similar investigation in Wolfram's book [14], Chapter 12, Section 8.

9.2.2 Resources

There are $(2sk)^{sk}$ s-state k-color Turing machines. That means 4 096 in (2,2) and 2 985 984 TMs in (3,2). In short, the number of TMs grows exponentially in the amount of resources. Thus, in representing our data and conventions we should be as economical as possible in using our resources so that exhaustive search in the smaller spaces still remains feasible. For example, an additional halting state will immediately increase the search space [1].

9.2.3 One-sided Turing Machines

In our experiment we have chosen to work with one-sided TMs. That is to say, we work with TMs with a tape that is unlimited to the left but limited to the right-hand side. One sided TMs are a common convention in the literature just perhaps slightly less common than the two sided convention. The following considerations led us to work with one-sided TMs.

- Efficient (that is, non-unary) number representations are place sensitive. That is to say, the interpretation of a digit depends on the position where the digit is in the number. Like in the decimal number 121, the leftmost 1 corresponds to the centenaries, the 2 to the decades and the rightmost 1 to the units. On a one-sided tape which is unlimited to the left, but limited on the right, it is straight-forward how to interpret a tape content that is almost everywhere zero. For example, the tape ...00101 could be interpreted as a binary string giving rise to the decimal number 5. For a two-sided infinite tape one can think of ways to come to a number notation, but all seem rather arbitrary.

1. Although in this case not exponentially so, as halting states define no transitions.

- With a one-sided tape there is no need for an extra halting state. We say that a computation simply halts whenever the head "drops off" the tape from the right hand side. That is, when the head is on the extremal cell on the right hand side and receives the instruction to moves right. A two-way unbounded tape would require an extra halting state which, in the light of considerations in 9.2.2 is undesirable.

On the basis of these considerations, and the fact that some work has been done before in the lines of this experiment [14] that also contributed to motivate our own investigation, we decided to fix the TM formalism and choose the one-way tape model.

9.2.4 Unary input representation

Once we had chosen to work with TMs with a one-way infinite tape, the next choice is how to represent the input values of the function. When working with two colors, there are basically two choices to be made: unary or binary. However, there is a very subtle point if the input is represented in binary. If we choose for a binary representation of the input, the class of functions that can be computed is rather unnatural and very limited.

The main reason is as follows. Suppose that a TM on input x performs some computation. Then the TM will perform the very same computation for any input that is the same as x on all the cells that were visited by the computation. That is, the computation will be the same for an infinitude of other inputs thus limiting the class of functions very severely. On the basis of these considerations we decided to represent the input in unary. Moreover, from a theoretical viewpoint it is desirable to have the empty tape input different from the input zero, thus the final choice for our input representation is to represent the number x by $x+1$ consecutive 1's.

The way of representing the input in unary has two serious draw-backs:

1. The input is very homogeneous. Thus, it can be the case that TMs that expose otherwise very rich and interesting behavior, do not do so when the input consists of a consecutive block of 1's.
2. The input is lengthy so that runtimes can grow seriously out of hand. See also our remarks on the cleansing process below.

We mitigate these objections with the following considerations.

1. Still interesting examples are found. And actually a simple informal argument using the Church-Turing thesis shows that universal functions will live in a canonical way among the thus defined functions.
2. The second objection is more practical and more severe. However, as the input representation is so homogeneous, often the runtime sequences exhibit so much regularity that missing values that are too large can be guessed. We shall do so as described in Subsection 9.2.7.

9.2.5 Binary output convention

None of the considerations for the input conventions applies to the output convention. Thus, it is wise to adhere to an output convention that reflects as much information about the final tape-configuration as possible. Clearly, by interpreting the output as a binary string, from the output value the output tape configuration can be reconstructed. Hence, our outputs, if interpreted, will be so as binary numbers.

Definition *[Tape Identity] We say that a TM computes the* tape identity *when the tape configuration at the end of a computation is identical to the tape configuration at the start of the computation.*

The output representation can be seen as a simple operation between systems, taking one representation to another. The main issue is, how does one keep the structure of a system when represented in another system, such that, moreover, no additional essential complexity is introduced.

For the tape identity, for example, one may think of representations that, when translated from one to another system, preserve the simplicity of the function. However, a unary input convention and a binary output representation immediately endows the tape identity with an exponential growth rate. In principle this need not be a problem. However, computations that are very close to the tape identity will give rise to number theoretic functions that are seemingly very complex. However, as we shall see, in our current set-up there will be few occasions where we actually do interpret the output as a number other than for representational purposes. In most of the cases the raw tape output will suffice.

9.2.6 The Halting Problem and Rice's theorem

By the Halting Problem and Rice's theorem we know that it is in general undecidable to know wether a function is computed by a particular TM

and whether two TMs define the same function. The latter is the problem of extensionality (do two TMs define the same function?) known to be undecidable by Rice's theorem. It can be the case that for TMs of the size considered in this paper, universality is not yet attained [2], that the Halting Problem is actually decidable in these small spaces and likewise for extensionallity.

As to the Halting Problem, we simply say that if a function does not halt after 1000 steps, it diverges. Theory tells that the error thus obtained actually drops exponentially with the size of the computation bound [6] and we re-affirmed this in our experiments too as is shown in Figure 9.2. After proceeding this way, we see that certain functions grow rather fast and very regular up to a certain point where they start to diverge. These obviously needed more than 1000 steps to terminate. We decided to complete these obvious non-genuine divergers manually. This process is referred to as *cleansing* and shall be addressed with more detail in the next subsection.

As to the problem of extensionality, we simply state that two TMs calculate the same function when they compute (after cleansing) the same outputs on the first 21 inputs 0 through 20 with a computation bound of 1000 steps. We found some very interesting observations that support this approach: for the (2,2) space the computable functions are completely determined by their behavior on the first 3 input values 0,1,2. For the $(3, 2)$ space the first 8 inputs were found to be sufficient to determine the function entirely.

9.2.7 Cleansing the data

As mentioned before, the Halting problem is undecidable so one will always err when mechanically setting a cut-off value for our computations. The choice that we made in this paper was as follows. We put the cut-off value at 1000. After doing so, we looked at the functions computed. For those functions that saw an initial segment with a very regular progression of runtimes, for example 16, 32, 64, 128, 256, 512, -1, -1, we decided to fill out the the missing values in a mechanized way. It is clear that, although better than just using a cut-off value, we will still not be getting all functions like this. Moreover, there is a probability that errors are made while filling out the missing values. However we deem the error not too significant, as we have a uniform approach in this process of filling out, that is, we apply the

2. Recent work ([17]) has shown some small two-way infinite tape universal TMs. It is known that there is no universal machine in the space of two-way unbounded tape (2,2) Turing machines but there is known at least one weakly universal Turing machine in (2,3)[14] and it may be (although unlikely) the case that a weakly universal Turing machine in (3,2) exists.

same process for all sequences, either in (2,2) or in (4,2) etc. Moreover, we know from theory ([6]) that most TMs either halt quickly or never halt at all and we affirmed this experimentally in this paper. Thus, whatever error is committed, we know that the effect of it is eventually only marginally. In this subsection we shall describe the way we mechanically filled out the regular progressions that exceeded the computation bound.

We wrote a so-called predictor program that was fed incomplete sequences and was to fill out the missing values. The predictor program is based on the function `FindSequenceFunction`[3] built-in to the computer algebra system *Mathematica*. Basically, it is not essential that we used `FindSequenceFunction` or any other intelligent tool for completing sequences as long as the cleansing method for all TM spaces is applied in the same fashion. A thorough study of the cleansing process, its properties, adequacy and limitations is presented in [19]. The predictor pseudo-code is as follows:

1. Start with the finite sequence of integer values (with -1 values in the places the machine didn't halt for that input index).
2. Take the first n consecutive non-divergent (convergent) values, where $n \geq 4$ (if there is not at least a segment with 4 consecutive non divergent values then it gives up).
3. Call `FindSequenceFunction` with the convergent segment and the first divergent value.
4. Replace the first divergent value with the value calculated by evaluating the function found by `FindSequenceFunction` for that sequence position.
5. If there are no more -1 values stop otherwise trim the sequence to the next divergent value and go to 1.

This is an example of a (partial) completion: Let's assume one has a sequence (2, 4, 8, 16, -1, 64, -1, 257, -1, -1) with 10 values. The predictor returns: (2,

3. `FindSequenceFunction` takes a finite sequence of integer values $\{a_1, a_2, \ldots\}$ and retrieves a function that yields the sequence a_n. It works by finding solutions to difference equations represented by the expression `DifferenceRoot` in *Mathematica*. By default, `DifferenceRoot` uses early elements in the list to find candidate functions, then validates the functions by looking at later elements. `DifferenceRoot` is generated by functions such as `Sum`, `RSolve` and `SeriesCoefficient`, also defined in *Mathematica*. `RSolve` can solve linear recurrence equations of any recurring order with constant coefficients. It can also solve many linear equations (up to second recurring order) with non-constant coefficients, as well as many nonlinear equations. For more information we refer to the extensive online *Mathematica* documentation.

4, 8, 16, 32, 64, 128, 257, -1, -1) because up to 257 the sequence seemed to be 2^n but from 257 on it was no longer the case, and the predictor was unable to find a sequence fitting the rest.

The prediction function was constrained by 1 second, meaning that the process stops if, after a second of trying, no prediction is made, leaving the non-convergent value untouched. This is an example of a completed Turing machine output sequence. Given (3, 6, 9, 12, -1, 18, 21, -1, 27, -1, 33, -1) it is retrieved completed as (3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36). Notice how the divergent values denoted by -1 are replaced with values completing the sequence with the predictor algorithm based in *Mathematica's* `FindSequenceFunction`.

The prediction vs. the actual outcome

For a prediction to be called successful we require that the output, runtime and space usage sequences coincide in every value with the step-by-step computation (after verification). One among three outcomes are possible:

- Both the step-by-step computation and the sequences obtained with `FindSequenceFunction` completion produce the same data, which leads us to conclude that the prediction was accurate.
- The step-by-step computation produces a non-convergent value -1, meaning that after the time bound the step-by-step computation didn't produce any new convergent value that wasn't also produced by the FindSequeceFunction (which means that either the value to be produced requires a larger time bound, or that the `FindSequenceFunction` algorithm has failed, predicting a convergent value where it is actually divergent).
- The step-by-step computation produced a value that the `FindSequenceFunction` algorithm did not predict.

In the end, the predictor indicated what machines we had to run for larger runtimes in order to complete the sequences up to a final time bound of 200 000 steps for a subset of machines that couldnt be fully completed with the predictor program. The number of incorrectly predicted (or left incomplete) in (3,2) was 90 out of a total 3368 sequences completed with the predictor program. In addition to these 45 cases of incorrect completions, we found 108 cases where the actual computation produced new convergent values that the predictor could not predict. The completion process led us to only eight final non-completed cases, all with super fast growing values.

In (4,2) things werent too different. Among the 30 955 functions that were sampled motivated by the functions computed in (3,2) that were found to have also been computed in (4,2) (having in mind a comparison of time complexity classes) only 71 cases could not be completed by the prediction process, or were differently computed by the step-by-step computation. That is only 0.00229 of the sequences, hence in both cases allowing us to make accurate comparisons with low uncertainty in spite of the Halting Problem and the problem of very large (although rare) halting times.

9.2.8 Running the experiment

To explore the different spaces of TMs we wrote a TM simulator in the programming language C. We tested this C language simulator against the `TuringMachine` function in *Mathematica* as it used the same encoding for TMs. It was checked and found in concordance for the whole (2,2) space and a sample of the (3,2) space.

We run the simulator in the cluster of the CICA (Centro de Informática Científica de Andalucía[4]). To explore the (2,2) space we used only one node of the cluster and it took 25 minutes. The output was a file of 2 MB. For (3,2) we used 25 nodes (50 microprocessors) and took a mean of three hours in each node. All the output files together fill around 900 MB.

9.3 Investigating the space of 2-state, 2-color Turing machines

In this section we shall have our first glimpse into the fascinating microcosmos of small Turing machines. We shall see what kind of computational behavior is found among the functions that live in (2,2) and reveal various complexity-related properties of the (2,2) space.

Definition *In our context and in the rest of this paper, an* algorithm *computing a function is one particular set of 21 quadruples of the form*

$$\langle \textit{input value}, \textit{output value}, \textit{runtime}, \textit{space usage} \rangle$$

for each of the input values $0, 1, \ldots, 20$*, where the output, runtime and space-usage correspond to that particular input.*

4. Andalusian Centre for Scientific Computing: `http://www.cica.es/`.

In the cleansed data of (2,2) we found 74 functions and a total of 138 different algorithms computing them.

9.3.1 Determinant initial segments

An indication of the complexity of the (2,2) space is the number of inputs needed to determine a function. In the case of (2,2) this number of inputs is only 3. For the first input, the input 0, there are 11 different outputs. The following list shows these different outputs (first value in each pair) and the frequency they appear with (second value in each pair). Output `-1` represents the divergent one:

```
{{3, 13}, {2, 12}, {-1, 10}, {0, 10}, {1, 10}, {7, 6}, {6, 4},
 {15, 4}, {4, 2}, {5, 2}, {31, 1}}
```

For two inputs there are 55 different combinations and for three we find all the 74 functions. The first input is most significant; without it, the other inputs only appear in 45 different combinations. This is because there are many functions with different behavior for the first input than for the rest.

We find it interesting that only 3 values of a TM are needed to fully determine its behavior in the full (2,2) space that consists of 4 096 different TMs. Just as a matter of analogy we bring the $\mathbf{C}^\infty$ functions to mind. These infinitely often differentiable continuous functions are fully determined by the outputs on a countable set of input values. It is an interesting question how the minimal number of input values needed to determine a TM grows relative to the total number of $(2 \cdot s \cdot k)^{s \cdot k}$ many different TMs in (s, k) space, or relative to the number of defined functions in that space.

9.3.2 Halting probability

In the cumulative version of Figure 9.2 we see that more than 63% of executions stop after 50 steps, and little growth is obtained after more steps. Considering that there is an amount of TMs that never halt, it is consistent with the theoretical result in [6] that most TMs stop quickly or never halt.

Let us briefly comment on Figure 9.2. First of all, we stress that the halting probability ranges over all pairs of TMs in (2,2) and all inputs between 0 and 20. Second, it is good to realize that the graph is some sort of best fit and leaves out zero values in the following sense. It is easy to see that on

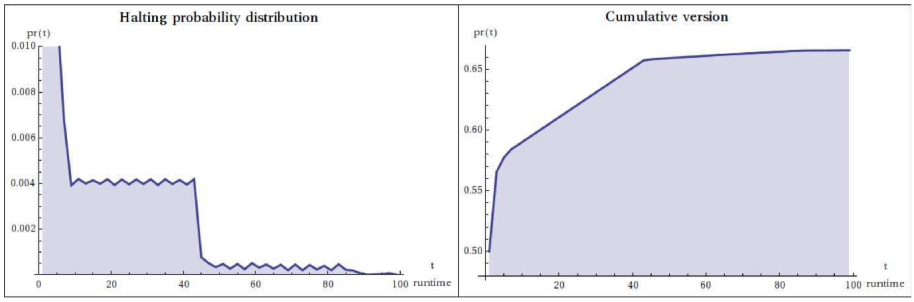

Figure 9.2: Halting times in (2,2).

the one-sided TM halting can only occur after an odd number of steps. Thus actually, the halting probability of every even number of steps is zero. This is not so reflected in the graph because of a smooth-fit.

We find it interesting that Figure 9.2 shows features reminiscent of phase transitions. Completely contrary to what we would have expected, these "phase transitions" were even more pronounced in $(3, 2)$ as one can see in Figure 9.12.

9.3.3 Phase transitions in the halting probability distribution

Let consider Figure 9.2 again. Note that in this figure only pairs of TMs and inputs are considered that halt in at most 100 steps. The probability of stopping (a random TM in $(2, 2)$ with a random input in 0 to 20) in at most 100 steps is 0.666. The probability of stopping in any number of steps is 0.667, so most TMs stop quickly of do not stop.

We clearly observe a phase transition phenomenon. To investigate the cause of this, let us consider the set of runtimes and the number of their occurrences. Figure 9.3 shows at the left the 50 smallest runtimes and the number of occurrences in the space that we have explored. The phase-transition is apparently caused because there are some blocks in the runtimes. To study the cause of this phase-transition we should observe that the left diagram on Figure 9.3 represents the occurrences of runtimes for arbitrary inputs from 0 to 20. The graph on the right of Figure 9.3 is clearer. Now, lines correspond to different inputs from 0 to 20. The graph at the left can be obtained from the right one by adding the occurrences corresponding to points with the same runtime. The distribution that we observe here explains

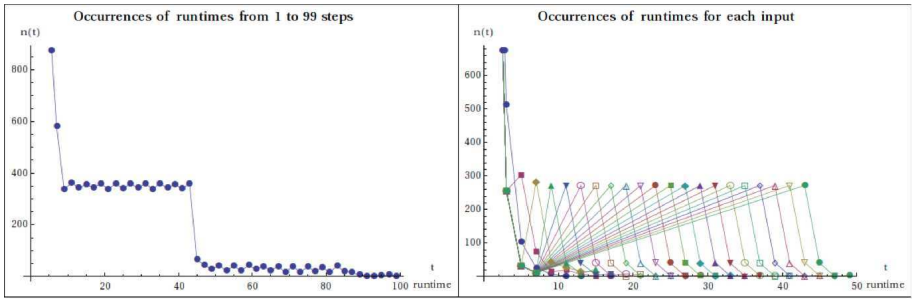

Figure 9.3: Occurrences of runtimes

the phase-transition effect. It's very interesting that in all cases there is a local maximum with around 300 occurrences and after this maximum, the evolution is very similar. In order to explain this, we look at the following list [5] that represents the 10 most frequent runtime sequences in $(2, 2)$. Every runtime sequence is preceded by the number of TMs computing it:

2048 {1, 1, ...}	106 {-1, 3, 3, ...}	20 {3, 7, 11, 15, ...}
1265 {-1, -1, ...}	76 {3, -1, -1, ...}	20 {3, 5, 5, ...}
264 {3, 5, 7, 9, ...}	38 {5, 7, 9, 11, ...}	
112 {3, 3, ...}	32 {5, 3, 3, ...}	

We observe that there are only 5 sequences computed more than 100 times. They represent 92.65% of the TMs in $(2, 2)$. There is only one sequence that is not constant nor divergent (recall that -1 represents divergences) with 264 occurrences: $\{3, 5, 7, 9, \ldots\}$. That runtime sequence corresponds to TMs that give a walk forth and back over the input tape to run of the tape and halt. This is the most trivial linear sequence and explains the intermediate step in the phase-transition effect. There is also another similar sequence with 38 occurrences $\{5, 7, 9, 11, \ldots\}$. Moreover, observe that there is a sequence with 20 occurrences where subsequent runtimes differ by 4 steps. This sequence $\{3, 7, 11, 15, \ldots\}$ contains alternating values of our original one $\{3, 5, 7, 9, \ldots\}$ and it explains the zigzag observed in the left part of Figures 9.2 and 9.3.

Altogether, this analysis accounts for the observed phase transition. In a sense, the analysis reduces the phase transition to the strong presence of linear performers in Figure 9.18 together with the facts that on the one hand there are few different kinds of linear performers and on the other hand that each group of similar linear TMs is "spread out over the horizontal axis" in Figure 9.2 as each input 0,..., 20 is taken into account.

5. The dots denote a linear progression (or constant which is a special case of linear).

9.3.4 Runtimes

There is a total of 49 different sequences of runtimes in (2,2). This number is 35 when we only consider total functions. Most of the runtimes grow linear with the size of the input. A couple of them grow quadratically and just two grow exponentially. The longest halting runtime occurs in TM numbers 378 and 1351, that run for 8 388 605 steps on the last input, that is on input 20. Both TMs used only 21 cells [6] for their computation and outputted the value 2 097 151.

Rather than exposing lists of outputvalues we shall prefer to graphically present our data. The sequence of output values is graphically represented as follows. On the top line we depict the tape output on input zero (that is, the input consisted of just one black cell). On the second line immediately below the first one, we depict the tape output on input one (that is, the input consisted of two black cells), etc. By doing so, we see that the function computed by TM 378 is just the tape identity.

Let us focus on all the (2,2) TMs that compute that tape identity. We will depict most of the important information in one overview diagram. This diagram as shown in Figure 9.4 contains at the top a graphical representation of the function computed as described above.

Below the representation of the function, there are six graphs. On each horizontal axis of these graphs, the input is plotted. The τ_i is a diagram that contains plots for all the runtimes of all the different algorithms computing the function in question. Likewise, σ_i depicts all the space-usages occurring. The $<\tau>$ and $<\sigma>$ refer to the (arithmetical) average of time and space usage. The subscript h in e.g. $<\tau>_h$ indicates that the harmonic average is calculated. As the harmonic average is only defined for non-zero numbers, for technical reasons we depict the harmonic average of $\sigma_i + 2$ rather than for σ_i.

Let us recall a definition of the harmonic mean. The harmonic mean of n non-zero values $x_1, \ldots, x_n$ is defined as

$$<x>_h := \frac{n}{\frac{1}{x_1} + \ldots + \frac{1}{x_n}}.$$

In our case, the harmonic mean of the runtimes can be interpreted as follows. Each TM computes the same function. Thus, the total amount of information in the end computed by each TM per input is the same although runtimes

6. It is an interesting question how many times each cell is visited. Is the distribution uniform over the cells? Or centered around the borders?

may be different. Hence the runtime of one particular TM on one particular input can be interpreted as time/information. We now consider the following situation:

Let the exhaustive list of TMs computing a particular function f be $\{TM_1, \ldots, TM_n$ with runtimes t_1,
$\ldots, t_n\}$. If we normalize the amount of information computed by f to 1, we can interpret e.g. $\frac{1}{t_k}$ as the amount of information computed by TM_k in one time step. If we now let TM_1 run for 1 time unit, next TM_2 for 1 time unit and finally TM_n for 1 time unit, then the total amount of information of the output computed is $1/t_1 + ... + 1/t_n$. Clearly,

$$\overbrace{\frac{1}{<\tau>_h} + \ldots + \frac{1}{<\tau>_h}}^{n \text{ times}} = \frac{\overbrace{\frac{1}{t_1} + \ldots + \frac{1}{t_n}}^{n \text{ times}} + \ldots \frac{\frac{1}{t_1} + \ldots + \frac{1}{t_n}}{n}}{n} = \frac{1}{t_1} + \ldots + \frac{1}{t_n}.$$

Thus, we can see the harmonic average as the time by which the typical amount of information is gathered on a random TM that computes f. Alternatively, the harmonic average $<\tau>_h$ is such that $\frac{1}{<\tau>_h}$ is the typical amount of information computed in one time step on a random TM that computes f.

9.3.5 Clustering in runtimes and space-usages

Observe the two graphics in Figure 9.5. The left one shows all the runtime sequences in (2,2) and the right one the used-space sequences. Divergences are represented by -1, so they explain the values below the horizontal axis. We find some exponential runtimes and some quadratic ones, but most of them remain linear. All space usages in (2,2) are linear.

An interesting feature of Figure 9.5 is the clustering. For example, we see that the space usage comes in three different clusters. The clusters are also present in the time graphs. Here the clusters are less prominent as there are more runtimes and the clusters seem to overlap. It is tempting to think of this clustering as rudimentary manifestations of the computational complexity classes.

Another interesting phenomenon is observed in these graphics. It is that of alternating divergence, detected in those cases where value -1 alternates with the other outputs, spaces or runtimes. The phenomena of alternating divergence is also manifest in the study of definable sets.

The image provides the basic information of the TM outputs depicted by a diagram with each row the output of each of the 21 inputs, followed by the plot figures of the average resources taken to compute the function, preceded by the time and space plot for each of the algorithm computing the function. For example, this info box tells us that there are 1 055 TMs computing the identity function, and that these TMs are distributed over just 12 different algorithms (i.e. TMs that take different space/time resources). Notice that at first glance at the runtimes τ_i, they seem to follow just an exponential sequence while space grows linearly. However, from the other diagrams we learn that actually most TMs run in constant time and space. Note that all TMs that run out of the tape in the first step without changing the cell value (the 25% of the total space) compute this function.

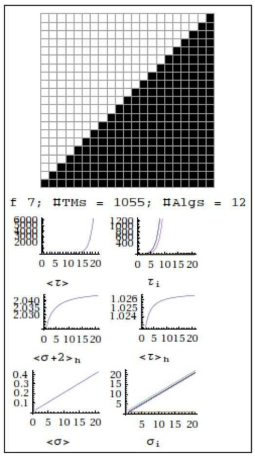

Figure 9.4: Overview diagram of the tape identity.

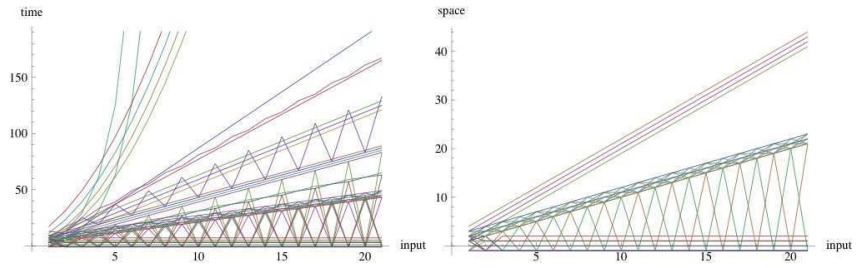

Figure 9.5: Runtime and space distribution in (2,2).

9.3.6 Definable sets

Like in classical recursion theory, we say that a set W is definable by a (2,2) TM if there is some machine M such that $W = W_M$ where W_M is defined as usual as

$$W_M := \{x \mid M(x) \downarrow\}.$$

In total, there are 8 definable sets in (2,2). Below follows an enumeration of them.

```
{{}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20}, {0}, {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20},
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20}, {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20}, {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}, {0, 1}}
```

It is easy to see that the definable sets are closed under complements.

9.3.7 Clustering per function

We have seen that all runtime sequences in (2,2) come in clusters and likewise for the space usage. It is an interesting observation that this clustering also occurs on the level of single functions. Some examples are reflected in Figure 9.6.

9.3.8 Computational figures reflecting the number of available resources

Certain functions clearly reflect the fact that there are only two available states. This is particularly noticeable from the period of alternating converging and non-converging values and in the offset of the growth of the output,

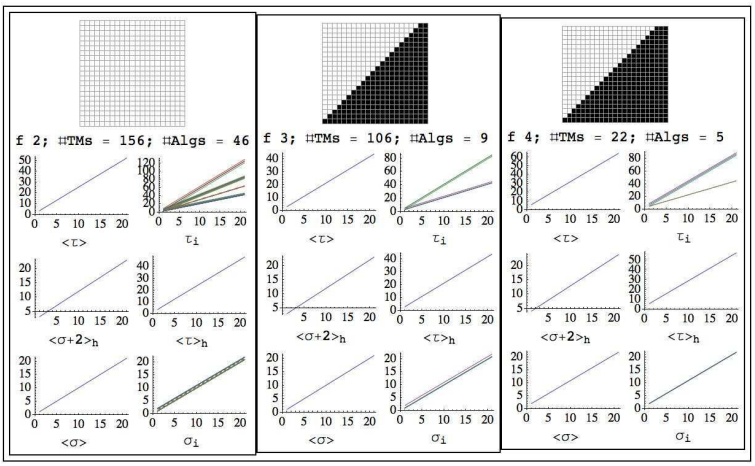

Figure 9.6: Clustering of runtimes and space-usage per function.

and in the alternation period of black and white cells. Some examples are included in Figure 9.7.

9.3.9 Types of computations in (2,2)

Let us finish this analysis with some comments about the computations that we can find in (2,2). Most of the TMs perform very simple computations. Apart from the 50% that in every space finishes the computations in just one step (those TMs that move to the right from the initial state), the general pattern is to make just one round through the tape and back. It is the case for TM number 2240 with the sequence of runtimes:

```
{5, 5, 9, 9, 13, 13, 17, 17, 21, 21, ..}
```

Figure 9.8 shows the sequences of tape configurations for inputs 0 to 5. Each of these five diagrams should be interpreted as follows. The top line represents the tape input and each subsequent line below that represents the tape configuration after one more step in the computation.

The walk around the tape can be more complicated. This is the case for TM number 2205 with the runtime sequence:

```
{3, 7, 17, 27, 37, 47, 57, 67, 77, ...}
```

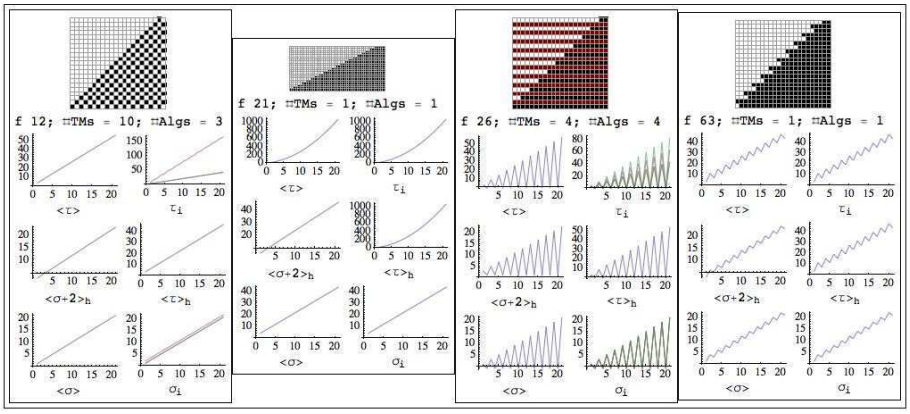

Figure 9.7: Computational figures reflecting the number of available resources.

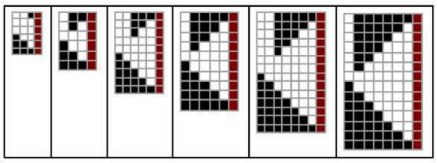

Figure 9.8: Turing machine tape evolution for Rule 2240.

which has a greater runtime but it only uses that part of the tape that was given as input, as we can see in the computations (Figure 9.9, left). TM 2205 is interesting in that it shows a clearly localized and propagating pattern that contains the essential computation.

The case of TM 1351 is one of the few that escapes from this simple behavior. As we saw, it has the highest runtimes in (2,2). Figure 9.9 (right) shows its tape evolution. Note that it is computing the tape identity. Many other TMs in (2,2) compute this function in linear or constant time. In this case of TM 1351 the pattern is generated by a genuine recursive process thus explaining the exponential runtime.

In (2,2) we also witnessed TMs performing iterative computations that gave rise to mainly quadratic runtimes. An example of this is TM 1447, whose computations for the first seven inputs are represented in Figure 9.10.

Let us briefly summarize the types of computations that we saw in (2,2).

- Constant time behavior like the head (almost) immediately dropping off the tape;
- Linear behavior like running to the end of the tape and then back again

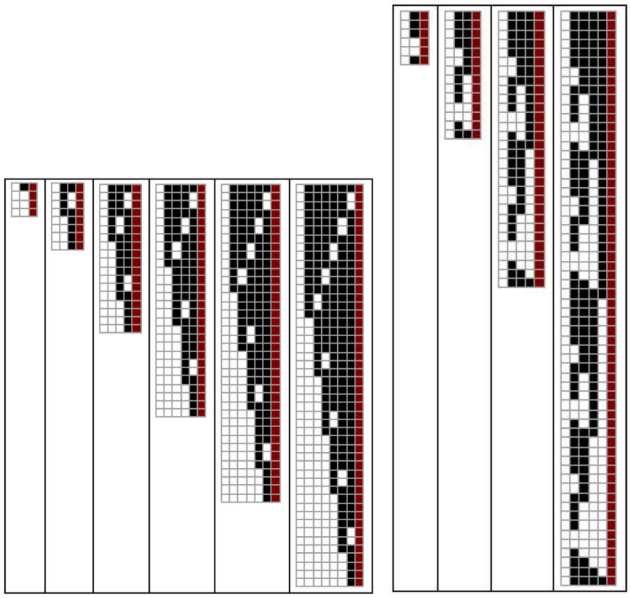

Figure 9.9: Tape evolution for Rules 2205 (left) and 1351 (right).

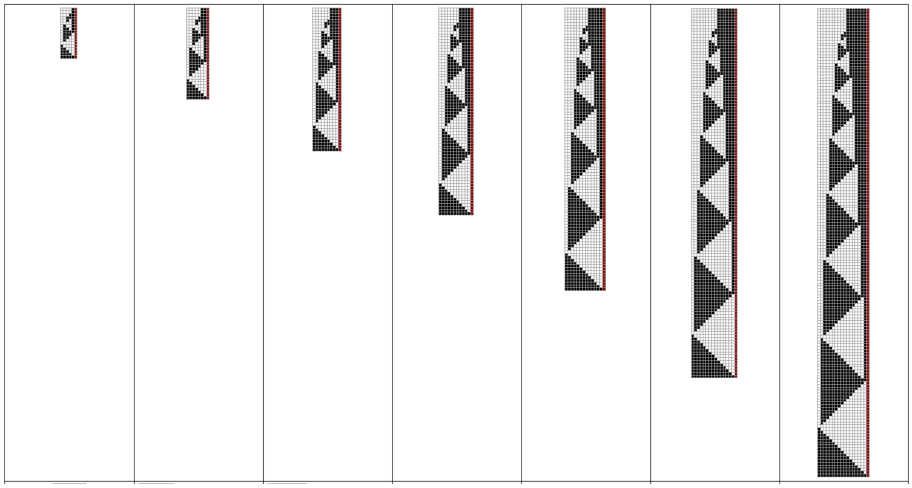

Figure 9.10: Turing machine tape evolution for Rule 1447.

as Rule 2240;
- Iterative behavior like using each black cell to repeat a certain process as in Rule 1447;
- Localized computation like in Rule 2205;
- Recursive computations like in Rule 1351.

As most of the TMs in (2,2) compute their functions in the easiest possible way (just one crossing of the tape), no significant speed-up can be expected. Only slowdown is possible in most cases.

9.4 Investigating the space of 3-states, 2-colors Turing machines

In the cleansed data of (3,2) we found 3886 functions and a total of 12824 different algorithms that computed them.

9.4.1 Determinant initial segments

As these machines are more complex than those of (2,2), more outputs are needed to characterize a function. From 3 required in (2,2) we need now 8, see Figure 9.11.

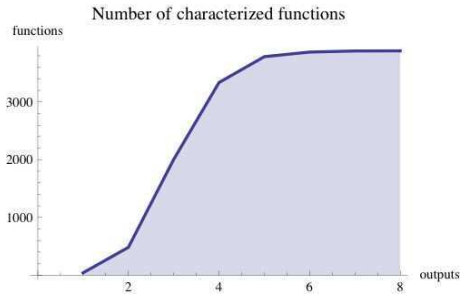

Figure 9.11: Number of outputs required to characterize a function in (3,2).

9.4.2 Halting probability

Figure 9.12 shows the runtime probability distributions in (3,2). The same behavior that we commented for (2,2) is also observed.

Note that the "phase transitions" in (3,2) are even more pronounced than in (2,2). We can see these phase transitions as rudimentary manifestations of computational complexity classes. Similar reasoning as in Subsection 9.3.3 can be applied for (3,2) to account for the phase transitions as we can see in Figure 9.13.

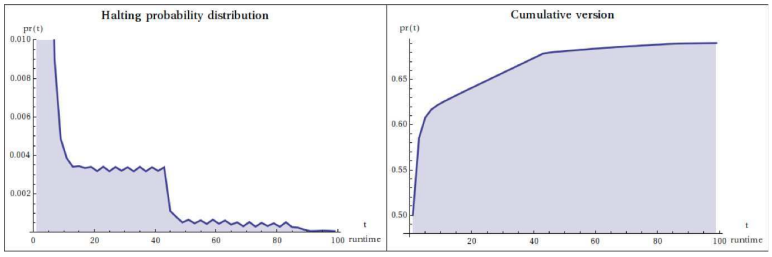

Figure 9.12: Runtime proprobability distributions in (3,2).

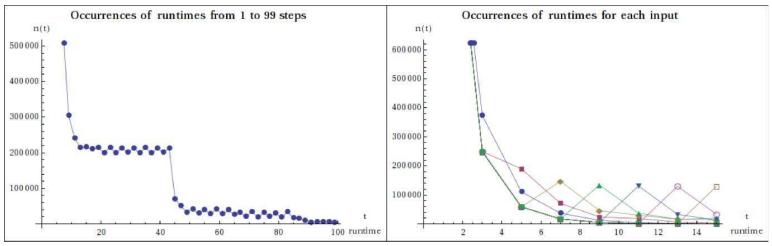

Figure 9.13: Occurrences of runtimes

9.4.3 Runtimes and space-usages

In (3,2) the number of different runtimes and space usage sequences is the same: 3676. Plotting them all as we did for (2,2) would not be too informative in this case. So, Figure 9.14 shows samples of 50 sequences of space and runtime sequences. Divergent values are omitted as to avoid big sweeps in the graphs caused by the alternating divergers. As in (2,2) we observe the same phenomenon of clustering.

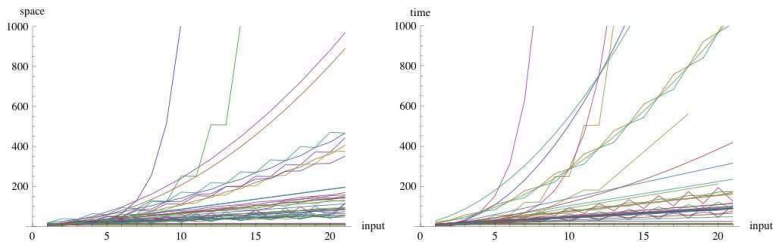

Figure 9.14: Sampling of 50 space (left) and runtime (right) sequences in (3,2).

9.4.4 Definable sets

Now we have found 100 definable sets. Recall that in (2,2) definable sets were closed under taking complements. This does not happen in (3,2). There are 46 definable sets, like

```
{{}, {0}, {1}, {2}, {0, 1}, {0, 2}, {1, 2}, {0, 1, 2}, ...}
```

that coexist with their complements, but another 54, like

```
{{0, 3}, {1, 3}, {1, 4}, {0, 1, 4}, {0, 2, 3}, {0, 2, 4}, ...}
```

are definable sets but their complements are not. We note that, although there are more definable sets in (3,2) in an absolute sense, the number of definable sets in (3,2) relative to the total amount of functions in (3,2) is about four times smaller than in (2,2).

9.4.5 Clustering per function

In (3,2) the same phenomenon of the clustering of runtime and space usage within a single function also happens. Moreover, as Figure 9.15 shows,

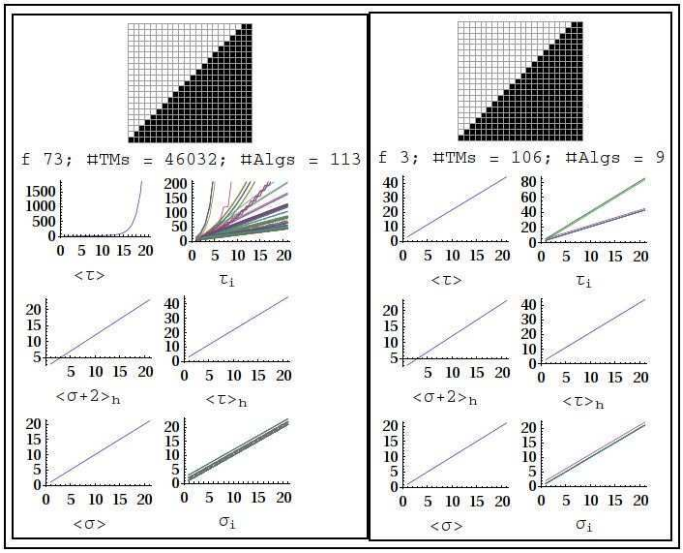

Figure 9.15: Clustering per function in (3,2).

exponential runtime sequences may occur in a (3,2) function (left) while only linear behavior is present among the (2,2) computations of the function (right).

9.4.6 Exponential behavior in (3,2) computations

Recall that in (2,2) most convergent TMs complete their computations in linear time. Now (3,2) presents more interesting exponential behavior, not only in runtime but also in used space.

The max runtime in (3,2) is 894 481 409 steps found in the TMs number 599063 and 666364 (a pair of twin rules[7]) at input 20. The values of this function are double exponential. All of them are a power of 2 minus 2.

Figure 9.16 shows the tape evolution with inputs 0 and 1. The pattern observed on the right repeats itself.

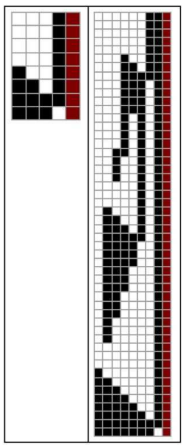

Figure 9.16: Tape evolution for Rule 599063.

9.5 The space (4,2)

An exhaustive search of this space fell out of the scope of the current project. For the sake of our investigations we were merely interested in finding functions in (4,2) that we were interested in. Thus, we sampled and looked only for interesting functions that we selected from (2,2) and (3,2). In searching the 4,2 space, we proceeded as follows. We selected 284 functions in (3,2), 18 of them also in (2,2), that we hoped to find in (4,2) using a sample of about $56 \text{x} 10^6$ random TMs.

7. We call two rules in (3,2) *twin rules* whenever they are exactly the same after switching the role of State 2 and State 3.

Our search process consisted of generating random TMs and run them for 1000 steps, with inputs from 0 to 21. The output (with runtime and space usage) was saved only for those TMs with a converging part that matches some of the 284 selected functions.

We saved 32235683 TMs. From these, 28 032 552 were very simple TMs that halt in just one step for every input, so we removed them. We worked with 4 203 131 non-trivial TMs.

After cleansing there were 1549 functions computed by 49 674 algorithms. From these functions, 22 are in (2,2) and 429 in (3,2). TMs computing all the 284 functions of the sampling were found.

Throughout the remainder of the paper it is good to constantly have in mind that the sampling in the (4,2) space is not at all representative.

9.6 Comparison between the TM spaces

The most prominent conclusion from this section is that when computing a particular function, slow-down of a computation is more likely than speed-up if the TMs have access to more resources to perform their computations. Actually no essential speed-up was witnessed at all. We shall compare the runtimes both numerically and asymptotically.

9.6.1 Runtimes comparison

In this section we compare the types of runtime progressions we encountered in our experiment. We use the big $\mathcal{O}$ notation to classify the different types of runtimes. Again, it is good to bear in mind that our findings are based on just 21 different inputs. However, the estimates of the asymptotic behavior is based on the functions as found in the cleansing process and sanity checks on more inputs confirmed the correctness (plausibility) of those functions.

Below, a table is presented that compares the runtime behavior between functions that were present in (2,2), (3,2) and (4,2). The first column refers to a canonical index for this list of functions that live in all of (2,2), (3,2) and (4,2). The column under the heading (2,2) displays the distribution of time complexity classes for the different algorithms in (2,2) computing the particular function in that row and likewise for the columns (3,2) and (4,2). Each time complexity class is followed by the number of occurrences

#	(3,2)	(4,2)
1	$O[n], 1265$ $O\lceil n^2 \rceil, 7$ $O\lceil n^3 \rceil, 1$	$O[n], 23739$ $O\lceil n^2 \rceil, 80$ $O\lceil n^3 \rceil, 6$
2	$O[n], 82$ $O\lceil n^2 \rceil, 1$	$O[n], 1319$ $O\lceil n^2 \rceil, 28$
4	$O[n], 133$ $O\lceil n^2 \rceil, 2$	$O[n], 2764$ $O\lceil n^2 \rceil, 72$ $O\lceil n^3 \rceil, 1$
6	$O[1], 23$ $O[n], 34$ $O\lceil n^2 \rceil, 9$ $O\lceil n^3 \rceil, 15$ $O\lceil n^4 \rceil, 5$ $O[\text{Exp}], 15$	$O[1], 197$ $O[n], 377$ $O\lceil n^2 \rceil, 101$ $O\lceil n^3 \rceil, 181$ $O\lceil n^4 \rceil, 59$ $O[\text{Exp}], 156$
10	$O[n], 54$ $O\lceil n^2 \rceil, 4$	$O[n], 502$ $O\lceil n^2 \rceil, 23$ $O[\text{Exp}], 4$
12	$O\lceil n^2 \rceil, 11$	$O\lceil n^2 \rceil, 110$ $O\lceil n^3 \rceil, 1$ $O[\text{Exp}], 3$
13	$O[n], 63$ $O\lceil n^2 \rceil, 7$ $O[\text{Exp}], 4$	$O[n], 544$ $O\lceil n^2 \rceil, 54$ $O\lceil n^3 \rceil, 16$ $O[\text{Exp}], 26$
32	$O\lceil n^2 \rceil, 12$	$O\lceil n^2 \rceil, 112$ $O\lceil n^3 \rceil, 3$ $O[\text{Exp}], 5$
95	$O[1], 8$ $O[n], 7$ $O\lceil n^3 \rceil, 1$ $O[\text{Exp}], 3$	$O[1], 49$ $O[n], 63$ $O\lceil n^2 \rceil, 15$ $O\lceil n^3 \rceil, 24$ $O\lceil n^4 \rceil, 4$ $O[\text{Exp}], 29$
100	$O[n], 9$ $O\lceil n^2 \rceil, 1$	$O[n], 90$ $O\lceil n^2 \rceil, 4$
112	$O[n], 5$ $O\lceil n^2 \rceil, 1$ $O\lceil n^3 \rceil, 3$	$O[n], 41$ $O\lceil n^2 \rceil, 10$ $O\lceil n^3 \rceil, 12$
135	$O\lceil n^2 \rceil, 13$	$O\lceil n^2 \rceil, 107$ $O\lceil n^3 \rceil, 4$ $O[\text{Exp}], 1$
138	$O[n], 5$ $O\lceil n^2 \rceil, 3$ $O\lceil n^3 \rceil, 1$	$O[n], 34$ $O\lceil n^2 \rceil, 13$ $O\lceil n^3 \rceil, 10$
292	$O[n], 7$ $O\lceil n^2 \rceil, 1$	$O[n], 162$ $O\lceil n^2 \rceil, 5$
350	$O[n], 1$ $O\lceil n^2 \rceil, 1$	$O[n], 20$ $O\lceil n^2 \rceil, 1$ $O\lceil n^3 \rceil, 2$
421	$O\lceil n^2 \rceil, 11$	$O\lceil n^2 \rceil, 88$ $O\lceil n^3 \rceil, 1$ $O[\text{Exp}], 2$
422	$O\lceil n^2 \rceil, 2$	$O\lceil n^2 \rceil, 17$

Figure 9.17: Comparison of the distributions of time classes of algorithms computing a particular function for a sample of 17 functions computed both in (3,2) and (4,2). The function number is an index from the list containing all 429 functions considered by us, that were computed in both TM spaces.

among the algorithms in that TM space. The complexity classes are sorted in increasing order. Note that we only display a selection of the functions, but our selection is representative for the whole (2,2) space.

#	(2,2)	(3,2)	(4,2)
1	O[1] : 46 O[n] : 46	O[1] : 1109 O[n] : 1429 O $\lceil n^2 \rceil$: 7 O $\lceil n^3 \rceil$: 1	O[1] : 19298 O[n] : 28269 O $\lceil n^2 \rceil$: 77 O $\lceil n^3 \rceil$: 6
2	O[1] : 5 O[n] : 5	O[1] : 73 O[n] : 64 O $\lceil n^2 \rceil$: 7 O[Exp] : 4	O[1] : 619 O[n] : 566 O $\lceil n^2 \rceil$: 53 O $\lceil n^3 \rceil$: 16 O[Exp] : 26
3	O[1] : 2 O[n] : 2	O[1] : 129 O[n] : 139 O $\lceil n^2 \rceil$: 2	O[1] : 2483 O[n] : 3122 O $\lceil n^2 \rceil$: 68 O $\lceil n^3 \rceil$: 1
4	O[1] : 16 O[n] : 5 O[Exp] : 3	O[1] : 124 O[n] : 34 O $\lceil n^2 \rceil$: 9 O $\lceil n^3 \rceil$: 15 O $\lceil n^4 \rceil$: 5 O[Exp] : 15	O[1] : 1211 O[n] : 434 O $\lceil n^2 \rceil$: 101 O $\lceil n^3 \rceil$: 181 O $\lceil n^4 \rceil$: 59 O[Exp] : 156
5	O[1] : 2 O[n] : 2	O[1] : 34 O[n] : 34	O[1] : 289 O[n] : 285 O $\lceil n^2 \rceil$: 8
6	O[1] : 3 O[n] : 3	O[1] : 68 O[n] : 74	O[1] : 576 O[n] : 668 O $\lceil n^2 \rceil$: 9 O $\lceil n^3 \rceil$: 3
7	O[1] : 10	O[1] : 54 O[n] : 8	O[1] : 368 O[n] : 94 O $\lceil n^3 \rceil$: 4 O[Exp] : 6
8	O[n] : 1 O $\lceil n^2 \rceil$: 1	O[n] : 13 O $\lceil n^2 \rceil$: 13	O[n] : 112 O $\lceil n^2 \rceil$: 107 O $\lceil n^3 \rceil$: 4 O[Exp] : 1
9	O[1] : 2 O[n] : 2	O[1] : 58 O[n] : 54 O $\lceil n^2 \rceil$: 4	O[1] : 503 O[n] : 528 O $\lceil n^2 \rceil$: 23 O[Exp] : 4
10	O[n] : 1 O $\lceil n^2 \rceil$: 1	O[n] : 11 O $\lceil n^2 \rceil$: 11	O[n] : 114 O $\lceil n^2 \rceil$: 110 O $\lceil n^3 \rceil$: 1 O[Exp] : 3
11	O[n] : 1 O $\lceil n^2 \rceil$: 1	O[n] : 11 O $\lceil n^2 \rceil$: 11	O[n] : 91 O $\lceil n^2 \rceil$: 88 O $\lceil n^3 \rceil$: 1 O[Exp] : 2
12	O[n] : 1 O $\lceil n^2 \rceil$: 1	O[n] : 12 O $\lceil n^2 \rceil$: 12	O[n] : 120 O $\lceil n^2 \rceil$: 112 O $\lceil n^3 \rceil$: 3 O[Exp] : 5
13	O[1] : 5 O[n] : 5	O[1] : 39 O[n] : 43	O[1] : 431 O[n] : 546 O $\lceil n^2 \rceil$: 1
14	O[1] : 4 O[n] : 4	O[1] : 14 O[n] : 14	O[1] : 119 O[n] : 121 O $\lceil n^2 \rceil$: 5 O $\lceil n^3 \rceil$: 1
15	O[1] : 2	O[1] : 11 O[n] : 1	O[1] : 69 O[n] : 15 O $\lceil n^2 \rceil$: 1 O[Exp] : 3
16	O[1] : 18	O[1] : 27 O[n] : 7 O $\lceil n^3 \rceil$: 1 O[Exp] : 3	O[1] : 233 O[n] : 63 O $\lceil n^2 \rceil$: 15 O $\lceil n^3 \rceil$: 24 O $\lceil n^4 \rceil$: 4 O[Exp] : 29
17	O[1] : 2 O[n] : 2	O[1] : 33 O[n] : 33	O[1] : 298 O[n] : 294 O $\lceil n^2 \rceil$: 2 O $\lceil n^3 \rceil$: 2
18	O[1] : 1 O[n] : 1	O[1] : 9 O[n] : 9	O[1] : 94 O[n] : 94
19	O[1] : 1 O[n] : 1	O[1] : 78 O[n] : 87 O $\lceil n^2 \rceil$: 1	O[1] : 1075 O[n] : 1591 O $\lceil n^2 \rceil$: 28
20	O[1] : 1 O[n] : 1	O[1] : 15 O[n] : 15	O[1] : 76 O[n] : 75 O $\lceil n^2 \rceil$: 1
21	O[1] : 1 O[n] : 1	O[1] : 21 O[n] : 21	O[1] : 171 O[n] : 173
22	O[1] : 1 O[n] : 1	O[1] : 14 O[n] : 14	O[1] : 203 O[n] : 203 O $\lceil n^2 \rceil$: 2 O[Exp] : 4

No essentially (different asymptotic behavior) faster runtime was found in (3,2) compared to (2,2). Thus, no speed-up was found other than by a linear factor as reported in Subsection (9.6.3). That is, no algorithm in (3,2) computing a function in (2,2) was essentially faster than the fastest algorithm computing the same function in (2,2). Amusing findings were Turing machines both in (2,2) and (3,2) computing the tape identify function in as much as exponential time. They are an example of machines spending all resources to compute a simple function. Another example is the constant function f(n) = 0 computed in $O(n^2)$, $O(n^3)$, $O(n^4)$ and even $O(Exp)$.

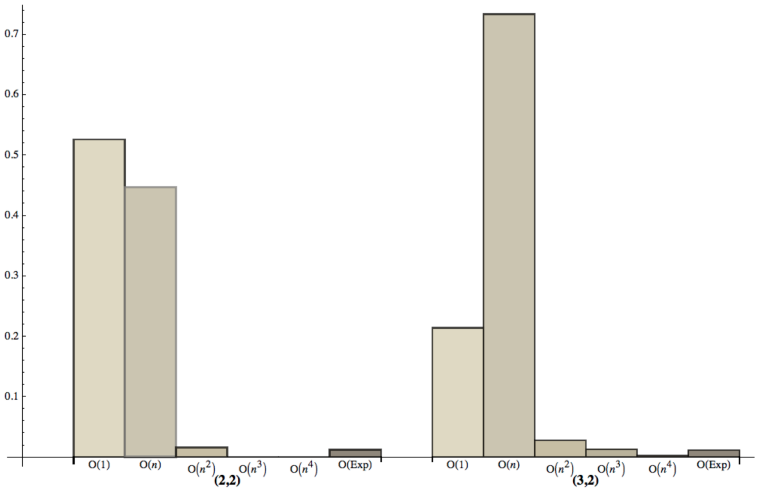

Figure 9.18: Time complexity distributions of (2,2) (left) and (3,2) (right).

In (2,2) however, there are very few non-linear time algorithms and functions[8]. However as we see from the similar table for (3,2) versus (4,2) in Figure 9.17, also between these spaces there is no essential speed-up witnessed. Again only speed-up by a linear factor can occur.

9.6.2 Distributions over the complexity classes

Figure 9.18 shows the distribution of the the TMs over the different asymptotic complexity classes. On the level of this distribution we see that the slow-down is manifested in a shift of the distribution to the right of the spectrum.

We have far to few data to possibly speak of a prior in the distributions of our TMs over these complexity classes. However, we do remark the following. In the following table we see the fraction per complexity class of the non-constant TMs for each space. Even though for (4,2) we do not at all work with a representative sampling still there is some similarity in the fractions. Most notably within one TM space, the ratio of one complexity class to another is in the same order of magnitude as the same ratio in one of the other spaces. Notwithstanding this being a far cry from a prior, we do find it worth[9] while mentioning.

8. We call a function $O(f)$ time, when its asymptotically fastest algorithm is $O(f)$ time.

9. Although we have very few data points we could still audaciously calculate the

pr	(2,2)	(3,2)	(4,2)
O(n)	0.941667	0.932911	0.925167
O(n^2)	0.0333333	0.0346627	0.0462362
O(n^3)	0	0.0160268	0.0137579
O(n^4)	0	0.0022363	0.00309552
O(Exp)	0.025	0.0141633	0.0117433

9.6.3 Quantifying the linear speed-up factor

For obvious reasons all functions computed in (2,2) are computed in (3,2). The most salient feature in the comparison of the (2,2) and (3,2) spaces is the prominent slowdown indicated by both the arithmetic and the harmonic averages. The space (3,2) spans a larger number of runtime classes. Figures 9.19 and 9.20 are examples of two functions computed in both spaces in a side by side comparison with the information of the function computed in (3,2) on the left side and the function computed by (2,2) on the right side. In [9] a full overview of such side by side comparison is published. Notice that the numbering scheme of the functions indicated by the letter f followed by a number may not be the same because they occur in different order in each of the (2,2) and (3,2) spaces but they are presented side by side for comparison with the corresponding function number in each space.

One important calculation experimentally relating descriptional (program-size) complexity and (time resources) computational complexity is the comparison of maximum of the average runtimes on inputs 0,...,20, and the estimation of the speed-ups and slowdowns factors found in (3,2) with respect to (2,2).

It turns out that 19 functions out of the 74 computed in (2,2) and (3,2) had at least one fastest computing algorithm in (3,2). That is a fraction of 0.256 of the 74 functions in (2,2). A further inspection reveals that among the 3 414 algorithms in (3,2), computing one of the functions in (2,2), only 122 were faster. If we supposed that "chances" of speed-up versus slow-down on the level of algorithms were fifty-fifty, then the probability that we observed at most 122 instantiations of speed-up would be in the order of 10^{-108}. Thus we can safely state that the phenomena of slow-down at the level of algorithms is significant.

Pearson coefficient correlations between the classes that are inhabited within one of the spaces. Among (2,2), (3,2) and (4,2) the Pearson coefficients are: 0.999737, 0.999897 and 0.999645.

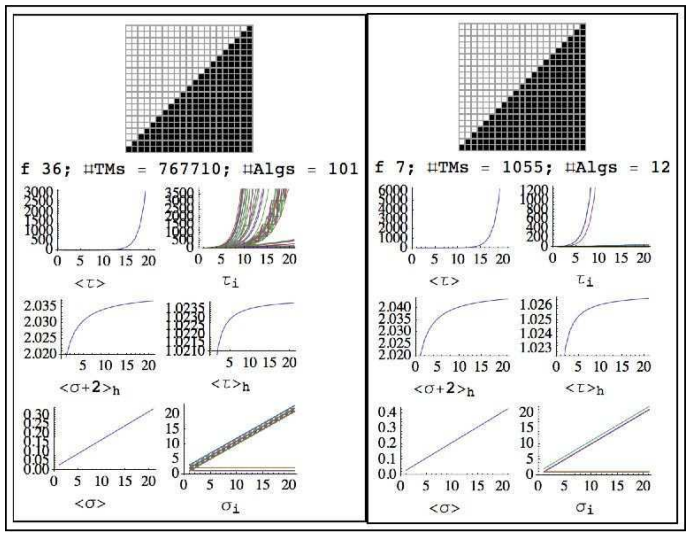

Figure 9.19: Side by side comparison of an example computation of a function in (2,2) and (3,2) (the identity function).

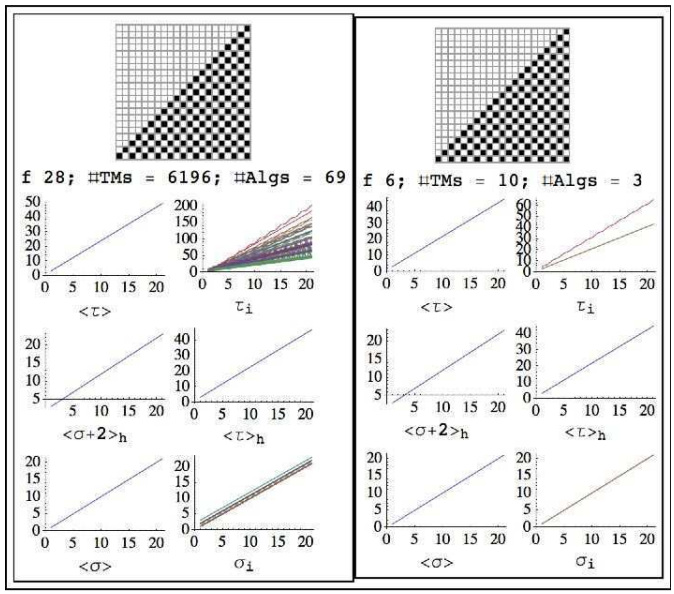

Figure 9.20: Side by side comparison of the computation of a function in (2,2) and (3,2).

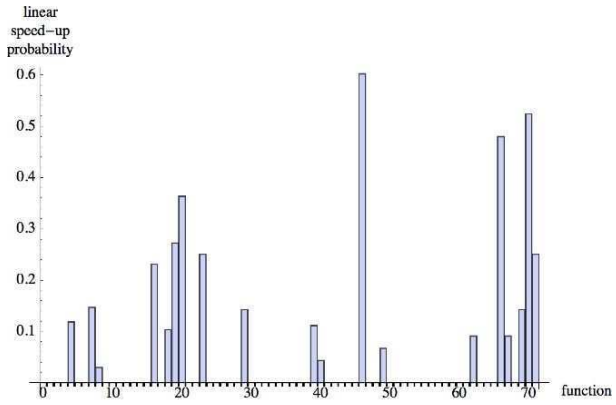

Figure 9.21: Distribution of speed-up probabilities per function. Interpreted as the probability of picking an algorithm in (3,2) computing faster an function in (2,2).

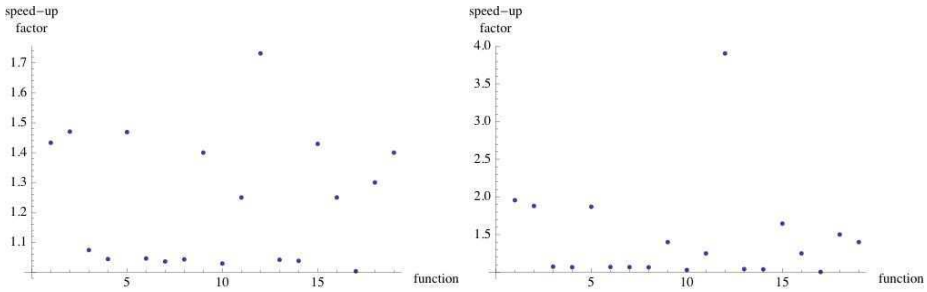

Figure 9.22: Speed up significance: on the left average and on the right maximum speed-ups.

Figure 9.23 shows the scarceness of the speed-up and the magnitudes of such probabilities. Figures 9.22 quantify the linear factors of speed-up showing the average and maximum. The typical average speed-up was 1.23 times faster for an algorithm found when there was a faster algorithm in (3,2) computing a function in (2,2).

In contrast, slowdown was generalized, with no speed-up for 0.743 of the functions. Slowdown was not only the rule but the significance of the slowdown was much larger than the scarce speed-up phenomenon. The average algorithm in (3,2) took 2 379.75 longer and the maximum slowdown was of the order of 1.19837×10^6 times slower than the slowest algorithm computing the same function in (2,2).

As mentioned before there is also no essential speed-up in the space (4,2) compared to (3,2) and only linear speed-up was witnessed at times. But

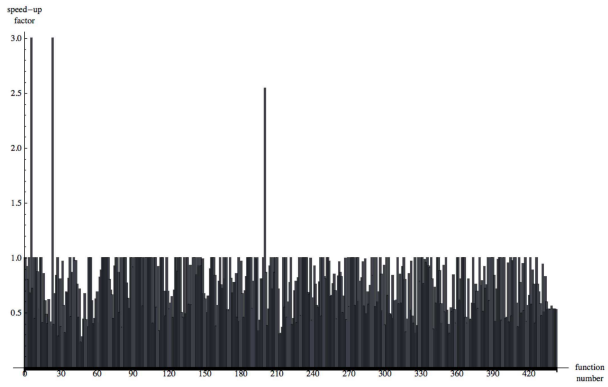

Figure 9.23: Distribution of average speed-up factors among all selected 429 functions computed in (3,2) and (4,2).

again, slow-down was the rule. Thus, (4,2) confirmed the trend between (2,2) and (3,2), that is that linear speed up is scarce yet present, three functions (0.0069) sampled from (3,2) had faster algorithms in (4,2) that in average took from 2.5 to 3 times less time to compute the same function, see Figure 9.6.3.

Acknowledgements

The initial motivation and first approximation of this project was developed during the NKS Summer School 2009 held at the Istituto di Scienza e Tecnologie dell'Informazione, CNR in Pisa, Italy. We wish to thank Stephen Wolfram for interesting suggestions and guiding questions. Furthermore, we wish to thank the CICA center and its staff for providing access to their supercomputing resources and their excellent support.

Bibliography

[1] C.H. Bennett. Logical Depth and Physical Complexity in Rolf Herken (ed), *The Universal Turing Machine–a Half-Century Survey*; Oxford University Press, p 227-257, 1988.

[2] C.H. Bennett. How to define complexity in physics and why, in *Complexity, entropy and the physics of information*, Zurek, W. H.; Addison-Wesley, Eds.; SFI studies in the sciences of complexity, p 137-148, 1990.

[3] M. Cook. Universality in Elementary Cellular Automata, *Complex Systems*, 2004.

[4] S. Cook. The complexity of theorem proving procedures, *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, p 151-158, 1971.

[5] G.J. Chaitin. *Gödel's theorem and information, Int. J. Theoret. Phys.*, 21, p 941-954, 1982.

[6] C.S. Calude, M.A. Stay, Most programs stop quickly or never halt, *Advances in Applied Mathematics*, 40, p 295-308, 2005.

[7] E. Fredkin. Digital Mechanics, *Physica D*, p 254-70, 1990.

[8] J. J. Joosten. Turing Machine Enumeration: NKS versus Lexicographical, *Wolfram Demonstrations Project*, 2010.

[9] J. J. Joosten, F. Soler Toscano, H. Zenil. Turing machine runtimes per number of states, to appear in *Wolfram Demonstrations Project*, 2011.

[10] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information and Transmission*, 1(1), p 1–7, 1965.

[11] L. Levin. Universal search problems, *Problems of Information Transmission* 9 (3), p 265-266, 1973.

[12] S. Lin & T. Rado. Computer Studies of Turing Machine Problems, *J. ACM*, 12, p 196-212, 1965.

[13] S. Lloyd, Programming the Universe; *Random House*, 2006.

[14] S. Wolfram, A New Kind of Science; *Wolfram Media*, 2002.

[15] Wolfram's 2, 3 Turing Machine Research Prize, http://www.wolframscience.com/prizes/tm23/; Accessed on June, 24, 2010.

[16] R. Neary, D. Woods. On the time complexity of 2-tag systems and small universal turing machines, In *FOCS*; IEEE Computer Society, p 439-448, 2006.

[17] D. Woods, T. Neary. Small semi-weakly universal Turing machines. *Fundamenta Informaticae*, 91, p 161-177, 2009.

[18] M. C. Wunderlich, A general class of sieve generated sequences, *Acta Arithmetica* 16, p 41-56, 1969.

[19] H. Zenil, F. Soler Toscano, J. J. Joosten. Empirical encounters with computational irreducibility and unpredictability, December 2010.