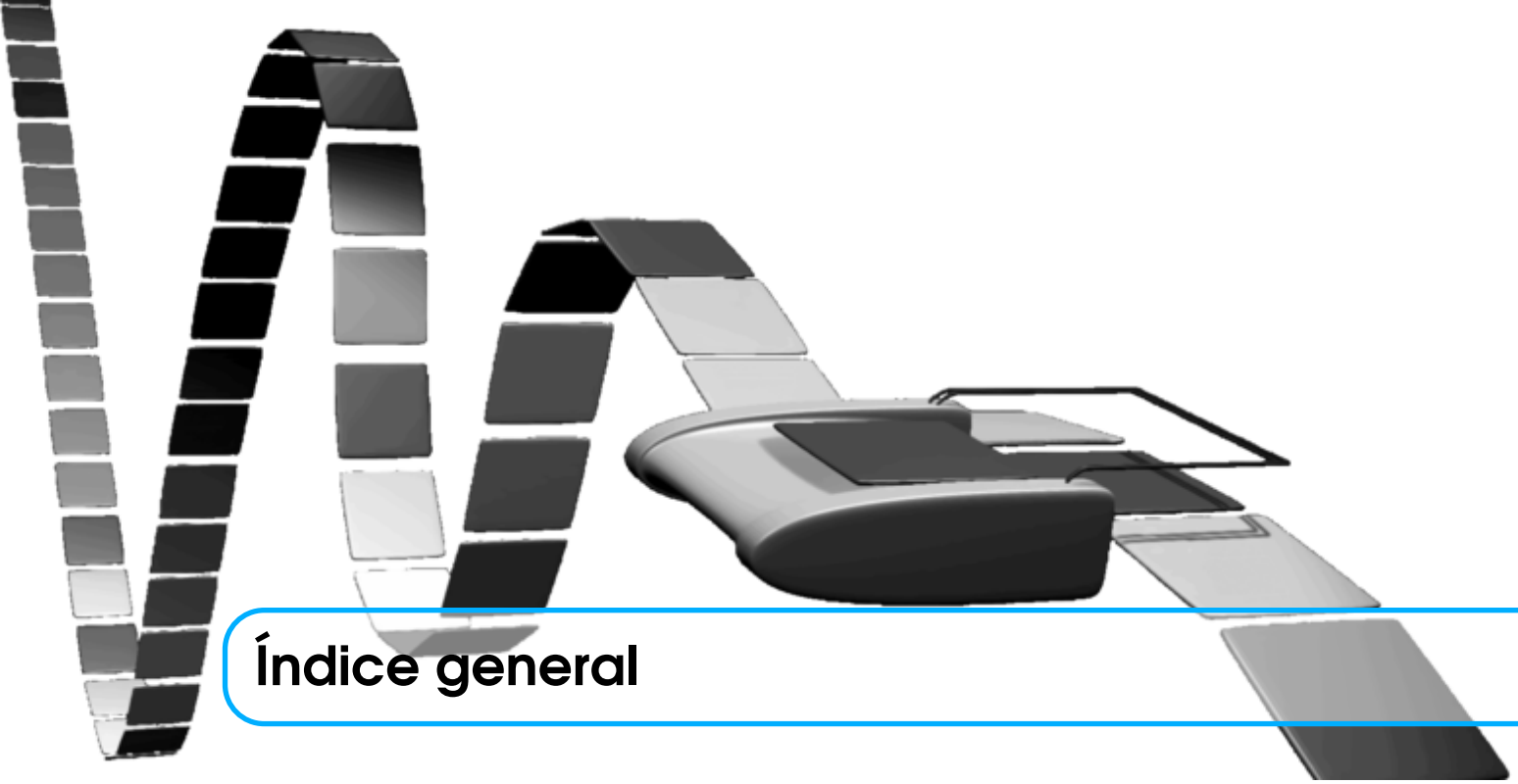
Teoría de la Información Algorítmica

Una guía práctica

Alejandro Puga Candelas Manuel de J. Luévano Robledo

PRIMERA EDICIÓN 2024

© 2024 Universidad Autónoma de Zacatecas “Francisco García Salinas”
Torre de Rectoría 3er Piso Campus UAZ Siglo XXI Carretera Zacatecas-Guadalajara km 6, Col. Ejido la Escondida
C.P 98000 Zacatecas, Zac. investigacionyposgrado@uaz.edu.mx
ISBN digital: 978-607-555-187-6 UAZ
La presente publicación pasó por el proceso de revisión de pares ciegos, bajo los criterios editoriales establecidos por parte del Programa Editorial de la Universidad Autónoma de Zacatecas.



Índice general

1 Prefacio 5
1.1 Sobre el lector 6
1.2 Sobre las herramientas computacionales 6

I Parte I: Teoría

2 ¿Cómo se mide la aleatoriedad? 11
2.1 Teoría de la Información y Entropía de Shannon 11
2.1.1 Entropía de bloque 12
2.2 Algoritmos de compresión 12
2.3 Complejidad Algorítmica 15
2.3.1 Probabilidad Algorítmica y Teorema de Codificación Algorítmica (CTM) 16

3 ¿Cómo aproximar la Complejidad Algorítmica? 17
3.1 CTM y el formalismo del Castor Atareado 17
3.2 Método de Descomposición en Bloques (BDM) 18
3.2.1 BDM vs Entropía de Shannon 19
3.3 Descomposición en bloques mediante particiones 19
3.3.1 Partición correlacionada 20
3.3.2 Partición recursiva 22
3.4 ¿Por qué funciona el BDM? (opcional) 23
3.4.1 Cotas superior e inferior del BDM 24
3.4.2 Convergencia del BDM hacia la entropía de Shannon en el peor de los casos 26

II Parte II: Implementación y Aplicaciones

4 Secuencias binarias 31
4.1 Introducción a la librería PyBDM 31
4.2 Cadenas binarias aleatorias 33
4.3 Secuencia Thue-Morse 36
4.4 Conclusiones 38

5 Arreglos bidimensionales y Grafos 41
5.1 Grafo ZK 41
5.2 Conclusiones 46

Bibliografía 49

1. Prefacio

En el tejido mismo de nuestra existencia, en cada parpadeo de nuestras percepciones, se revela un constante acto de computación. Nuestra mente, con una intrincada sofisticación, sigue un proceso algorítmico para desentrañar el tapiz del mundo que nos rodea. Desde la delicada captura de la luz por nuestros ojos hasta la sinfonía neural que interpreta esas señales, el acto de percibir y comprender se traduce en una sinfonía de operaciones, una coreografía de cálculos que podrían equipararse a un algoritmo complejo.

Este impulso de explicar lo que observamos ha sido un pilar fundamental en la evolución humana. En el afán de comprender, la ciencia encuentra su esencia más pura, reduciéndose en última instancia a la tarea esencial de clasificar la aparente aleatoriedad que nos envuelve. Cuando desentrañamos el mecanismo de un fenómeno natural, no solo estamos decodificando su aparente caos, sino descubriendo su orden subyacente, revelando que detrás de lo que parece caótico y caprichoso hay, de hecho, una causa subyacente. En este proceso de revelación, el fenómeno se transforma en un "objeto" abstracto, representado por una elegante cadena de símbolos que codifican su esencia.

Medir la aleatoriedad se ha convertido, entonces, en un desafío que ha intrigado a mentes inquisitivas a lo largo de la historia. A pesar de las diversas métricas desarrolladas por la comunidad científica, desde la venerada entropía de Shannon hasta los métodos de compresión, ninguna ha alcanzado la precisión inherente a la Complejidad Algorítmica (CA). La CA no es simplemente una herramienta; es un faro que ilumina las complejidades ocultas. Va más allá de proporcionarnos un medio para evaluar el contenido de un objeto; nos brinda un criterio preciso para discernir si un fenómeno es intrínsecamente aleatorio. Más aún, nos concede la capacidad única de encontrar una descripción más concisa que el propio fenómeno, desentrañando la esencia misma de la complejidad que se esconde en la aparente simplicidad.

Así, en las páginas de este libro, nos introducimos en el mundo de la Complejidad Algorítmica y su metodología.

1.1 Sobre el lector

Este libro es más que una guía convencional; es una invitación a la exploración de la Teoría de la Complejidad Algorítmica como una herramienta transformadora para científicos de diversas disciplinas. No es un libro de texto que busque sumergirse en los detalles más profundos de la metodología, sino más bien una brújula práctica para aquellos que desean incorporar esta poderosa perspectiva en su investigación.

Para aprovechar al máximo esta obra, se asume un conocimiento básico en teoría de probabilidad clásica, estructuras matemáticas, teoría de grafos y rudimentos de Ciencias de la Computación. Estos fundamentos proporcionan el andamiaje necesario para comprender y aplicar los conceptos que aquí se presentan. Este no es un impedimento, sino más bien un trampolín para científicos de cualquier campo que deseen adentrarse en la causalidad de los sistemas que observan y analizan.

El lenguaje Python será nuestro aliado en este viaje. Se espera que los lectores tengan un conocimiento básico-intermedio de Python (programación orientada a objetos, manejo de librerías, estructuras de datos, procesamiento y visualización de datos), ya que nos apoyaremos en la librería PyBDM, una herramienta valiosa que puede simplificar y potenciar la implementación de la Complejidad Algorítmica en sus investigaciones. También usaremos otras librerías tales, como Numpy, Matplotlib y Networkx.

A lo largo del libro encontrará el lector ejemplos concretos y aplicaciones prácticas que ilustran la implementación de la Complejidad Algorítmica en algunos contextos. Para facilitar la experiencia, hemos creado un repositorio en línea que contiene los códigos utilizados y las respuestas a los ejercicios propuestos. Este recurso adicional le permitirá no sólo entender los conceptos teóricos, sino también aplicarlos de manera práctica en sus propias investigaciones. El enlace al repositorio es `https://github.com/MJLRoad/AIT-guia-practica`.

Con estas herramientas a su disposición, este libro se convierte en un compañero confiable en su viaje hacia la comprensión más profunda y la aplicación efectiva de la Complejidad Algorítmica en su propia labor científica. ¡Bienvenido a este fascinante viaje de descubrimiento!

1.2 Sobre las herramientas computacionales

En este viaje hacia la complejidad algorítmica, nos apoyaremos brevemente en una herramienta excepcional: el *Online Algorithmic Complexity Calculator* (OACC) o Calculador Algorítmico en Línea, disponible en `http://complexity-calculator.com/index.html`. Esta potente herramienta proporciona estimaciones de la complejidad algorítmica, también conocida como la complejidad de Kolmogorov-Chaitin (o K), y la Probabilidad Algorítmica, tanto para cadenas cortas como largas, así como para matrices bidimensionales, de manera más precisa que cualquier otra herramienta disponible. Además, ofrece estimaciones de la Profundidad Lógica de Bennett (o LD) para cadenas, una medida que, hasta ahora, resultaba imposible de estimar con cualquier otra herramienta. Estas tres medidas se consideran las más poderosas y universales para evaluar la complejidad algorítmica.

Para estimar la complejidad de cadenas largas y matrices de adyacencia extensas, el OACC utiliza un método llamado BDM, basado en la Probabilidad Algorítmica. La estimación de complejidad por medio de BDM es compatible pero va más allá del alcance de los algoritmos de compresión sin pérdida, que son ampliamente utilizados para estimar K. A diferencia de los algoritmos de compresión sin pérdida, como LZ, LZW, DEFLATE, etc., que se basan exclusivamente en la entropía de Shannon (H), BDM no solo considera regularidades estadísticas, sino que también es sensible a segmentos de naturaleza algorítmica, tales como secuencias numéricas específicas. Estos últimos serían caracterizados por la entropía de Shannon y algoritmos de compresión sin pérdida

como de máxima aleatoriedad y el más alto grado de incompresibilidad. Además, a diferencia de la complejidad de Kolmogorov-Chaitin (gracias al Teorema de Invarianza), tanto la entropía como los algoritmos de compresión basados en entropía no son invariantes a la elección del lenguaje y, por tanto, no son lo suficientemente robustos para medir la complejidad o aleatoriedad.

Las herramientas para el lenguaje Python se pueden descargar de esta página web: `https://www.algorithmicdynamics.net/software.html`.

Para citar y reconocer el valor de esta herramienta, se solicita consultar la página oficial (`http://complexity-calculator.com/HowToCite.html`). Es importante destacar que el OACC es un proyecto desarrollado conjuntamente por los siguientes laboratorios: *Algorithmic Nature Group* y *Algorithmic Dynamics Lab*. Su colaboración ha dado lugar a una herramienta indispensable que enriquece nuestro viaje en el estudio de la complejidad algorítmica.

Parte I: Teoría

2 ¿Cómo se mide la aleatoriedad? 11
2.1 Teoría de la Información y Entropía de Shannon
2.2 Algoritmos de compresión
2.3 Complejidad Algorítmica

3 ¿Cómo aproximar la Complejidad Algorítmica? 17
3.1 CTM y el formalismo del Castor Atareado
3.2 Método de Descomposición en Bloques (BDM)
3.3 Descomposición en bloques mediante particiones
3.4 ¿Por qué funciona el BDM? (opcional)

2. ¿Cómo se mide la aleatoriedad?

En este capítulo hacemos una breve reseña de algunas métricas comunmente utilizadas para caracterizar la aleatoriedad en objetos. También introducimos la Complejidad Algorítmica o CA, su significado, y su conexión con la Probabilidad Algorítmica.

2.1 Teoría de la Información y Entropía de Shannon

La teoría de la información[Sha48] es una rama de las matemáticas que se ocupa de la cuantificación y comunicación de la información. Uno de los conceptos fundamentales de la teoría de la información es el concepto de *entropía*. En particular, la **entropía de Shannon** es una medida de la cantidad de incertidumbre o contenido de información en una variable aleatoria.

Definición 2.1.1 — Entropía de Shannon. La entropía de Shannon $H(X)$ de una variable aleatoria discreta X con función de probabilidad $p(x)$ se define como

$$H(x) = -\sum_{x \in \mathscr{X}} p(x) \log_2 p(x) \tag{2.1}$$

donde $\mathscr{X}$ es el conjunto de todos los posibles valores de X. El logaritmo normalmente se toma en base 2, por lo que la entropía se mide en bits.

La entropía de Shannon posee algunas propiedades importantes, tales como:

- Siempre es no negativa, $H(X) \geq 0$.
- Se maximiza cuando todos los posibles valores de X son igualmente probables, i.e., cuando $p(x) = 1/|\mathscr{X}|$ para todo $x \in \mathscr{X}$. En este caso, la entropía es $\log_2 |\mathscr{X}|$.
- Se minimiza cuando X es una función determinista de otra variable aleatoria Y, i.e., cuando existe una función f tal que $X = f(Y)$ con probabilidad 1.

En este último caso, la entropía es 0, ya que no existe incertidumbre o contenido de información en X más allá de la información contenida en Y.

La entropía de Shannon es una herramienta útil en una variedad de aplicaciones, incluida la teoría de codificación, la compresión de datos y la criptografía[Sto15]. La entropía de Shannon indica cuánta "sorpresa" hay en los resultados de una variable aleatoria. Cuanto mayor es la entropía, más incierta o impredecible es la variable aleatoria y más información se necesita para describir o comunicar sus resultados. Por el contrario, cuanto menor es la entropía, más segura o predecible es la variable aleatoria y menos información se necesita para describir o comunicar sus resultados.

■ **Ejemplo 2.1** Tenemos un dado profesional de alta precisión, como los que se usan en los casinos. Realizamos 1000 lanzamientos, y las frecuencias relativas obtenidas son $p(1) = 156/1000$, $p(2) = 162/1000$, $p(3) = 175/1000$, $p(4) = 179/1000$, $p(5) = 154/1000$ y $p(6) = 174/1000$. De la Ec. 2.1 tenemos que la entropía de la respectiva variable aleatoria es $H = 2.582$.

Repetimos el experimento, pero ahora con un dado normal, y obtenemos las siguientes frecuencias relativas: $p(1) = 167/1000$, $p(2) = 167/1000$, $p(3) = 162/1000$, $p(4) = 136/1000$, $p(5) = 159/1000$ y $p(6) = 209/1000$. Ahora la entropía es $H = 2.573$.

En comparación, la entropía de un dado perfectamente uniforme sería $H = \log_2 6 = 2.585$[1]. Vemos que la entropía del dado profesional es ligeramente más alta que la del dado normal. Esto se debe a que en el dado profesional cada número tiene casi la misma probabilidad de aparecer, lo que hace que sus resultados sean más impredecibles. Por otro lado, el dado normal muestra una mayor variabilidad en las frecuencias de los números, lo que hace que sus resultados sean un poco más predecibles. Por lo tanto, podemos concluir que el dado profesional es más 'justo' o equilibrado en términos de probabilidad de cada resultado. ■

■ **Ejemplo 2.2** Alicia lanza una moneda extraña 12 veces, y obtiene la siguiente serie de resultados (siendo 1 "águila" y siendo 0 "sello"): 101110110111. Las frecuencias relativas son $p(0) = 3/12$ y $p(1) = 9/12$. La entropía de la respectiva variable aleatoria es $H = 0.811$. Alicia concluye que la moneda está cargada, ya que es menos impredecible que una moneda "justa" con $H = 1.0$. ■

2.1.1 Entropía de bloque

Una advertencia con respecto a la entropía de Shannon es que nos vemos obligados a tomar una decisión arbitraria con respecto a la granularidad. Tomemos, por ejemplo, la cadena de bits 01010101010101. La entropía de Shannon de la cadena a nivel de bits individuales es máxima, pero la cadena es claramente regular si se toman bloques de dos bits (no superpuestos) como unidades básicas, en cuyo caso la cadena tiene una entropía mínima. Por lo tanto, una mejor versión de la entropía de Shannon puede ser reescrita como función de longitud de bloque, cuyo valor mínimo puede capturar la periodicidad de una cadena, vea Fig. 2.1.

> **Ejercicio 2.1** Elabore un script en Python (o cualquier otro lenguaje) para calcular y graficar la entropía de la secuencia `s_TM = 011010011001` en función del tamaño de bloque. Intente reproducir las primeras tres gráficas de la Fig. 2.1 para validar su script.
>
> (*Respuesta parcial:* El valor máximo de la entropía de bloque para `s_TM` es igual a 1.5) ■

2.2 Algoritmos de compresión

Se denomina **algoritmo de compresión sin pérdidas** a cualquier procedimiento de codificación que tenga como objetivo representar cierta cantidad de información utilizando u ocupando un

[1] Los datos provienen de este video corto: `https://www.youtube.com/shorts/ZmfECvxD_MA`

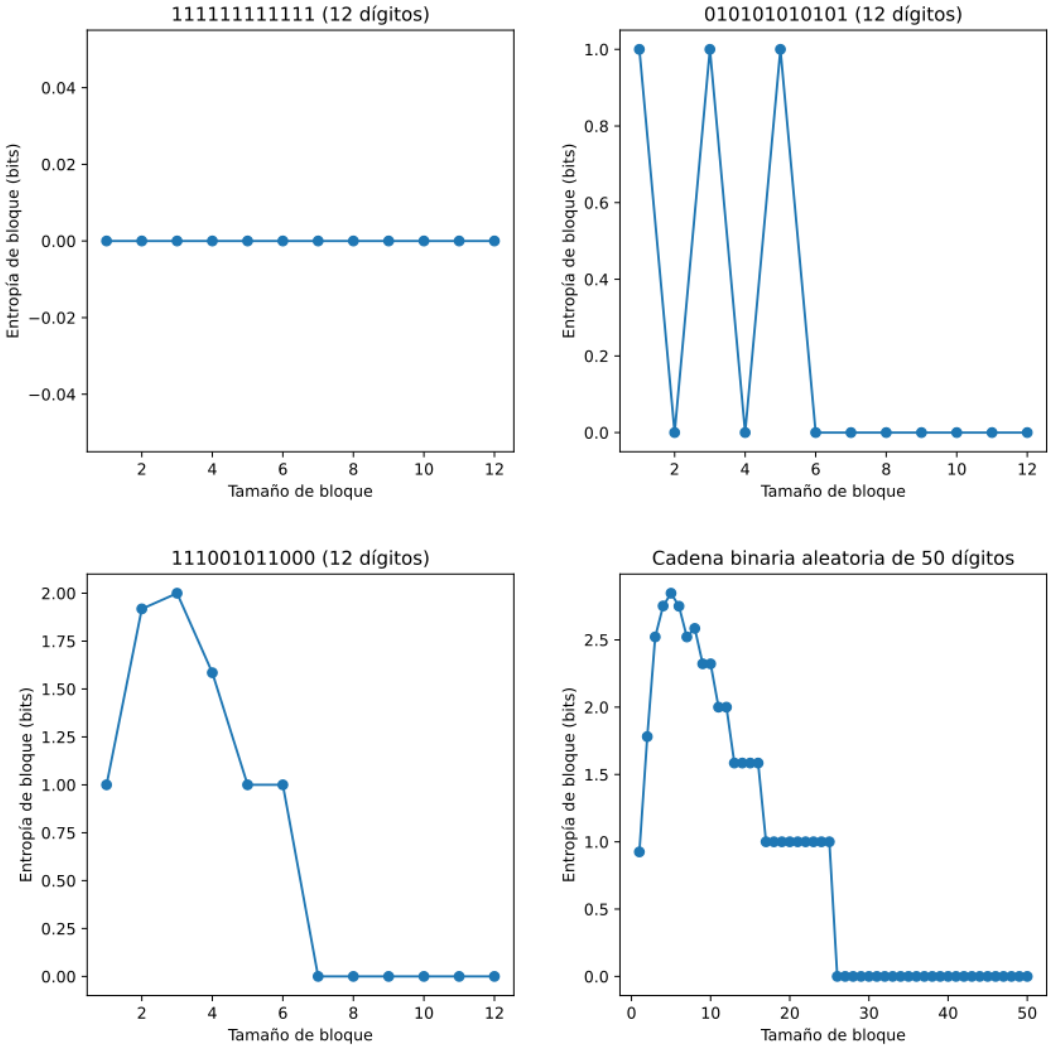


Figura 2.1: Entropía en función de tamaño de bloque para algunas cadenas binarias. Debido a que los bloques mayores que $n/2$ serían en efecto bloques únicos y por lo tanto tendrían una entropía igual a 0, el bloque más grande posible es de tamaño $n/2$.

Entrada	Salida	Siguiente símbolo	Agregar a diccionario
a	1	b	ab:4
b	2	a	ba:5
ab	4	c	abc:6
c	3	b	cb:7
ba	5	b	bab:8
bab	8	a	baba:9
a	1	a	aa:10
aa	10	a	aaa:11
aaa	11	a	aaaa:12
a	1	-	-

Cuadro 2.1: Codificación paso a paso de una cadena mediante el algoritmo LZW.

espacio menor, siendo posible una reconstrucción exacta de los datos originales. Un ejemplo es el **algoritmo de Lempel-Ziv-Welch (LZW)**[ZL78], un compresor sin pérdidas basado en diccionario. El algoritmo usa un diccionario para mapear la secuencia de símbolos a códigos que toman menos espacio. El algoritmo LZW se desempeña bien, especialmente con archivos que tienen secuencias repetitivas.

El algoritmo de codificación LZW es el siguiente:

1. Empezar con un diccionario de tamaño 256 o 4096 (donde la clave es un caracter ASCII y los valores son consecutivos).
2. Encontrar la clave W en el diccionario más grande (en valor) que coincida con los primeros caracteres de la cadena de entrada.
3. Dar como salida el valor de la clave W y remover W de la cadena de entrada.
4. Agregar (W+(El primer símbolo)) : ((tamaño de diccionario)+1) al diccionario.
5. Repetir los pasos 2 a 4 hasta que la cadena de entrada esté vacía.

Nótese que no es necesario almacenar el diccionario ya que el **proceso de descompresión LZW** reconstruye el diccionario a medida que avanza.

■ **Ejemplo 2.3** Tenemos una cadena de entrada `s = "ababcbababaaaaaaa"`. El diccionario inicial es `dict = {a:1, b:2, c:3}`. Implementamos el algoritmo de codificación LZW (vea Cuadro 2.1). Al final obtenemos la cadena codificada `sc = "124358110111"`. ■

Otro ejemplo de algoritmo de compresión sin pérdidas lo tenemos en el programa de compresión de archivos gratuito y de código abierto denominado **BZip2**, el cual utiliza el algoritmo de Burrows-Wheeler[Sny23], que solo comprime archivos individuales; no es un compilador de archivos. Se basa en utilidades externas independientes para tareas como el manejo de múltiples archivos, el cifrado y la división de archivos.

Ejercicio 2.2 Implemente el algoritmo de compresión LZW (puede ser de forma escrita o mediante un script) para comprimir las cadenas del Ejercicio 2.1,

- `s_a = "111111111111"`
- `s_b = "010101010101"`
- `s_c = "111001011000"`
- `s_TM = "011010011001"`

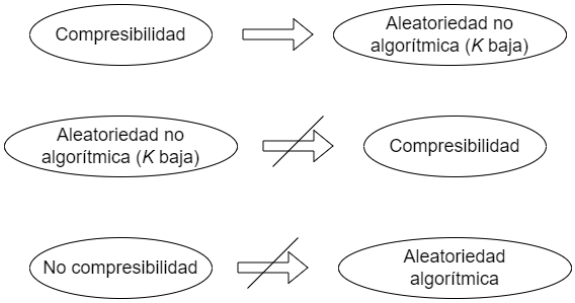


Figura 2.2: Condiciones necesarias y suficientes de (no) aleatoriedad algorítmica.

A partir de las cadenas codificadas, calcule la **razón de compresibilidad** -igual a (longitud de cadena codificada) / (longitud de cadena inicial)- de cada cadena.

(*Respuesta parcial*: Las razones de compresibilidad son 0.417, 0.5, 0.667 y 0.667). ■

2.3 Complejidad Algorítmica

Definida por Solomonoff, Kolmogorov, Chaitin y Levin, la complejidad del tamaño del programa, también conocida como **complejidad algorítmica** o complejidad de Kolmogorov, es una medida que cuantifica la aleatoriedad algorítmica, un tipo de aleatoriedad que es estrictamente más fuerte que la aleatoriedad estadística.

Definición 2.3.1 — Complejidad Algorítmica (CA). La complejidad algorítmica, que denotaremos por K, de una cadena s es la longitud del programa de computadora más corto p que se ejecuta en una **máquina de Turing** (MT) universal sin prefijos (o *prefix-free*) U que genera la cadena como salida y se detiene [Cha66; Kol68]:

$$K(s) = \min\{|p| : U(p) = s\} \tag{2.2}$$

La diferencia mayor o menor entre la longitud original y la longitud comprimida, es decir, la longitud de la MT, determina la complejidad de la cadena. Se dice que una cadena s es aleatoria si $K(s)$ (en bits) $\sim |s|$

Un inconveniente técnico es que K es semicalculable superior. En otras palabras, no existe un algoritmo eficaz que tome una cadena s como entrada y produzca el número $K(s)$ como salida. K es incalculable debido al problema de parada[Cop04], dado que no siempre se pueden encontrar los programas más cortos en un tiempo finito sin tener que ejecutar todos los programas de computadora, lo que significa tener que esperar una eternidad en caso de que nunca se detengan.

En la práctica se siguen enfoques pragmáticos, como el uso generalizado de algoritmos de compresión sin pérdidas, para aproximar K. La versión comprimida de un objeto es un límite superior de su complejidad algorítmica, lo que significa que la complejidad algorítmica real de la cadena no puede ser mayor que su longitud comprimida. Por lo tanto, encontrar una representación breve de un objeto comprimiéndolo es una prueba suficiente de no aleatoriedad. El problema, por supuesto, son los casos en los que no se encuentra ninguna versión comprimida. Como K no es computable, no podemos garantizar que la falta de una versión comprimida de un objeto signifique que dicho objeto no existe, por lo que no se puede decir mucho al respecto. Dada esta limitación, en general es más importante y significativamente más informativo descubrir que algo es comprimible (vea Fig. 2.2).

2.3.1 Probabilidad Algorítmica y Teorema de Codificación Algorítmica (CTM)

La probabilidad clásica de producción de una cadena de bits s entre todas las 2^n posibles cadenas de bits de longitud n está dada por $p(s) = 1/2^n$. El concepto de **probabilidad algorítmica** (también conocida como semimedida de Levin) reemplaza la producción aleatoria de resultados con la producción aleatoria de programas que producen un resultado. La probabilidad algorítmica de una cadena s es, por tanto, una medida que estima la probabilidad p de que un programa aleatorio produzca una cadena s cuando se ejecuta en una MT universal sin prefijos U.

Definición 2.3.2 — Probabilidad Algorítmica. La probabilidad algorítmica $m(s)$ de una cadena binaria s es una suma sobre todos los programas p (sin prefijos) con los cuales una MT universal sin prefijos U genera a s y se detiene. Esta definición reemplaza a n (la longitud de s) con la longitud del programa que produce a s:

$$m(s) = \sum_{p:U(p)=s} \frac{1}{2^{|p|}} \tag{2.3}$$

El **teorema de codificación algorítmica**, o CTM (*Coding Theorem Method*)[Lev74; Sol64] establece además la conexión entre $m(s)$ y $K(s)$,

Teorema 2.3.1 — Teorema de Codificación Algorítmica (CTM).

$$|-\log_2 m(s) - K(s)| < c \tag{2.4}$$

donde c es una constante fija, independiente de s.

Este teorema implica[CCS13] que la distribución de frecuencia de salida de programas de computadora aleatorios para aproximar $m(s)$ se puede convertir en estimaciones de $K(s)$ usando

$$K(s) = -\log_2(m(s)) + O(1) \tag{2.5}$$

3. ¿Cómo aproximar la Complejidad Algorítmica?

En este capítulo se trata el problema de cómo llevar a la práctica la estimación de la complejidad algorítmica en objetos de diverso tamaño, no obstante el alto coste computacional que aquello supondría debido al CTM.

3.1 CTM y el formalismo del Castor Atareado

El CTM se trata de aproximar la complejidad algorítmica de un objeto mediante la ejecución de todo programa posible, desde el más corto hasta el más largo, y contar el número de veces que un programa produce al objeto. Esto último conlleva una búsqueda exhaustiva sobre el conjunto infinito-contable de programas de computadora que se pueden escribir en un lenguaje o MT universal dada de referencia. Un programa de longitud n tiene una probabilidad asintótica cercana a 1 de detenerse después de 2^n pasos, lo que hace que este procedimiento sea exponencialmente costoso.

Para darle la vuelta a este problema, se puede llevar a cabo una búsqueda exhaustiva para un número de programas (o MT's) lo "suficientemente pequeño" en los cuales se conoce el tiempo de parada gracias al **problema del Castor Atareado**[Rad62]. En este problema, se propone encontrar la MT de tamaño fijo (estados y símbolos) que tarda más en correr respecto a otras máquinas del mismo tamaño. Se conocen valores para MT's de 2 símbolos y hasta 4 estados, que pueden ser usados para descartar cualquier MT que tome más pasos que los valores del Castor Atareado.

Sea (t,k) el conjunto de todas las MT con t estados y k símbolos -usando el formalismo del Castor Atareado, y sea $T \in (t,k)$ una MT con input vacío. La distribución de salida empírica para una secuencia s,

$$D(t,k)(s) = \frac{|\{T \in (t,k) : T \text{ produce a } s\}|}{|\{T \in (t,k) : T \text{ se detiene}\}|} \tag{3.1}$$

nos da una estimación de la probabilidad algorítmica de s. $D(t,k)$ es computable para valores pequeños de t y k, cuyos valores de castor atareado son conocidos.

Dentro de este formalismo, un Castor Atareado es la MT de t estados y k símbolos que escribe un número máximo de símbolos antes de parar, habiendo comenzado en una cinta en blanco. Sin embargo, debido al problema de parada, solo se puede computar el Castor Atareado para valores de t y k pequeños. Aun así, se puede continuar la aproximación de D para valores más grandes de t y k mediante un muestreo.

De este modo, la medida de complejidad por CTM es igual a

$$\mathrm{CTM}(s,t,k) = -\log_b D(t,k)(s) \tag{3.2}$$

con b siendo el número de símbolos en el alfabeto -tradicionalmente igual a 2 para objetos binarios.

3.2 Método de Descomposición en Bloques (BDM)

El Método de Descomposición en Bloques (BDM, por sus siglas en inglés) es una herramienta poderosa para medir la complejidad de sistemas complejos. Este método permite descomponer un sistema grande en piezas más pequeñas y manejables, facilitando así la estimación de su complejidad total. Al dividir los datos en fragmentos y precomputar cálculos de CTM sobre estos fragmentos, el BDM proporciona una medida que se encuentra entre la entropía y la complejidad algorítmica, permitiendo una mejor comprensión de la estructura y el comportamiento del sistema en estudio[Zen+18].

Definición 3.2.1 — BDM. Sea $D(t,k)(s)$ la distribución de frecuencias construída al correr todas las MTs con t estados y k símbolos. Definimos el BDM de la secuencia s como

$$\mathrm{BDM}(s,l,m) = \sum_i \left[\mathrm{CTM}\left(s^i,t,k\right) + \log_b n_i\right] \tag{3.3}$$

donde

- $s^i \equiv$ la subsecuencia i
- $l \equiv$ longitud de cada subsecuencia
- $n_i \equiv$ multiplicidad de subsecuencias

El método de descomposición en bloques se puede extender para objetos más allá de las cadenas unidimensionales, tales como arreglos que representan mapas de bits o grafos (matriz de adyacencia). Para ello, hay que considerar MT's bidimensionales.

En general, para objetos w-dimensionales, necesitaríamos primero los valores CTM de los objetos base. Más específicamente,

$$\mathrm{BDM}(X,\{x_i\}) = \sum_{(r_i,n_i)\in \mathrm{Adj}(X)_{\{x_i\}}} [\mathrm{CTM}(r_i) + \log_b n_i] \tag{3.4}$$

donde

- $r_i \equiv$ elemento de la descomposición de X (especificada por la partición $\{x_i\}$)
- $n_i \equiv$ multiplicidad de cada $\{r_i\}$
- $\mathrm{CTM}(r_i) \equiv$ aproximación computable de $\mathrm{K}(r_i)$ obtenida mediante la aplicación de CTM con MT's w-dimensionales.

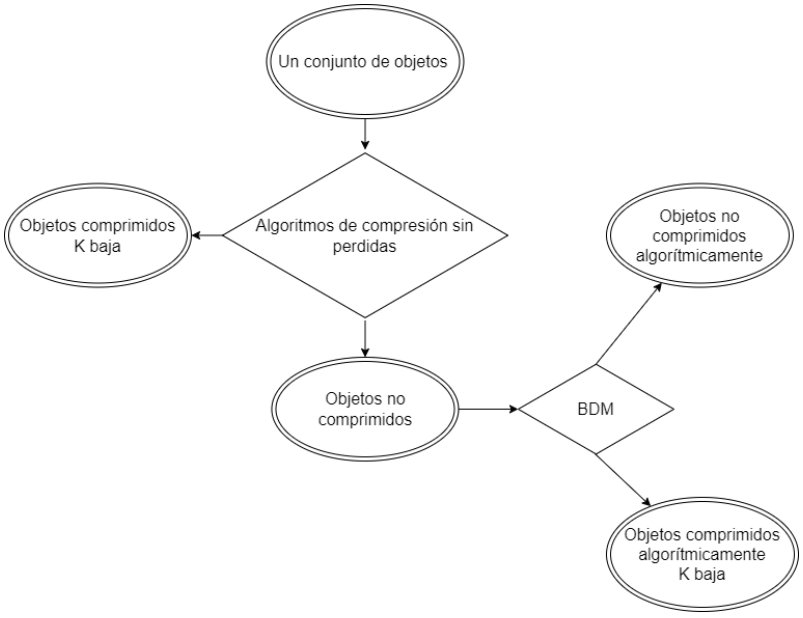


Figura 3.1: El BDM nos puede ayudar a encontrar objetos con aleatoriedad baja, más allá de lo que es posible con algoritmos de compresión.

3.2.1 BDM vs Entropía de Shannon

En el pasado, los algoritmos de compresión sin pérdidas dominaban el panorama de las aplicaciones de complejidad algorítmica. Cuando los investigadores optaron por utilizar algoritmos de compresión sin pérdidas para cadenas razonablemente largas, el método demostró ser valioso (por ejemplo, [CV05]). Su aplicación exitosa ha tenido que ver con el hecho de que la compresibilidad es una prueba suficiente para determinar la aleatoriedad no algorítmica (aunque lo contrario no es cierto). Sin embargo, las implementaciones populares de algoritmos de compresión sin pérdidas se basan en estimaciones de entropía[HH19] y, por lo tanto, no están más estrechamente relacionadas con la complejidad algorítmica que la entropía de Shannon en sí misma. Sólo pueden dar cuenta de regularidades estadísticas y no algorítmicas, aunque tener en cuenta las regularidades algorítmicas debería ser crucial, ya que estas regularidades representan la principal ventaja de utilizar la complejidad algorítmica.

Es de esperar que la mayoría de las cadenas tengan tanto entropía máxima -i.e., no son comprimibles- como complejidad algorítmica máxima -la mayoría de las cadenas binarias no se pueden relacionar con programas más cortos ya que estos también son cadenas binarias. Aun así, existe un número infinito de secuencias con entropía máxima y baja complejidad algorítmica. El BDM asigna menor complejidad a más cadenas que la entropía, como es de esperar. A diferencia de la entropía y las implementaciones de algoritmos de compresión sin pérdidas, el BDM reconoce algunas cadenas que no tienen regularidades estadísticas pero que tienen contenido algorítmico que las hace comprimibles algorítmicamente (vea Fig. 3.1).

3.3 Descomposición en bloques mediante particiones

Al hecer la partición de un objeto -cadena, arreglo o tensor- quedan “sobras” en la frontera, y las únicas dos opciones para tomarlas en cuenta en la estimación de la complejidad algorítmica del objeto son las siguientes:

- Definir una ventana deslizante que permita el traslape

- Estimar la contribución a la complejidad de las sobras.

Cada una de estas opciones conlleva un tipo específico de partición del objeto a analizar. A continuación veremos los dos métodos de partición, sus ventajas y desventajas a la hora de aplicarlos para estimar la complejidad algorítmica mediante BDM.

3.3.1 Partición correlacionada

Volviendo a la Definición 3.2.1 de BDM para una cadena unidimensional, puede haber una secuencia residual de longitud $|y| < |l|$ si $|s|$ no es múltiplo de l. En este caso se define una "ventana deslizable" con traslape igual a $l-m$, donde m es un parámetro de deslizamiento tal que

- m puede ir de 1 a l.
- $m = l$ significa que no hay traslape, o sea, que se tiene una partición exacta de s.
- $m < l$ significa que tenemos traslape, el cual va a influir en el valor de la estimación por BDM.

Entre más pequeño sea m, es decir, entre más grande sea el traslape, mayor será la sobre-estimación de BDM.

■ **Ejemplo 3.1** Tenemos la cadena binaria $s = 010101010101010101$, con $|s| = 18$. Para $l = 12$ y $m = 1$, s se descompone en las subsecuencias que se muestran en la Fig. 3.2. La subsecuencia $s^1 = 010101010101$ tiene multiplicidad $n_1 = 4$, y la subsecuencia $s^2 = 101010101010$ tiene multiplicidad $n_2 = 3$. Los valores CTM para esas secuencias son[DZ12; Sol+14]

$$\text{CTM}\left(s^1 = 010101010101, t = 2, k = 2\right) = 26.99073$$

$$\text{CTM}\left(s^2 = 101010101010, t = 2, k = 2\right) = 26.99073$$

y el BDM es igual a

$$\begin{aligned} \text{BDM}(s, l = 12, m = 1) &= \sum_i \left[\text{CTM}\left(s^i, t = 2, k = 2\right) + \log_2 n_i\right] \\ &= 26.99073 + \log_2 4 + 26.99073 + \log_2 3 \\ &= 57.566 \end{aligned} \tag{3.5}$$

(La base para el logaritmo de las multiplicidades es 2, porque el alfabeto de la cadena s tiene 2 caracteres, al igual que el número de símbolos $k = 2$ en el conjunto de MTs) ■

En general el traslape de bloques produce sobreestimaciones de complejidad (en el Ejemplo 3.1 se obtiene un valor nada despreciable de BDM no obstante que la cadena binaria es periódica), mientras que el no traslape da lugar a una subestimación solamente para objetos con sobras, es decir, objetos con dimensiones que no son múltiplos del tamaño de bloque. Se recomienda limitar el uso de la partición correlacionada por dos razones:

- El traslape da lugar a sobreestimaciones.
- En el no traslape, la subestimación resultante de omitir la evaluación de fronteras o sobras converge y está acotada por una fracción de la complejidad total.

Aun así, hay ocasiones en las que esta partición es preferible a una partición sin traslapes. Para ver por qué, se debe tener en cuenta el siguiente hecho:

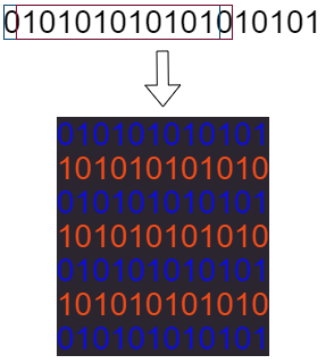


Figura 3.2: Partición correlacionada de una cadena binaria.

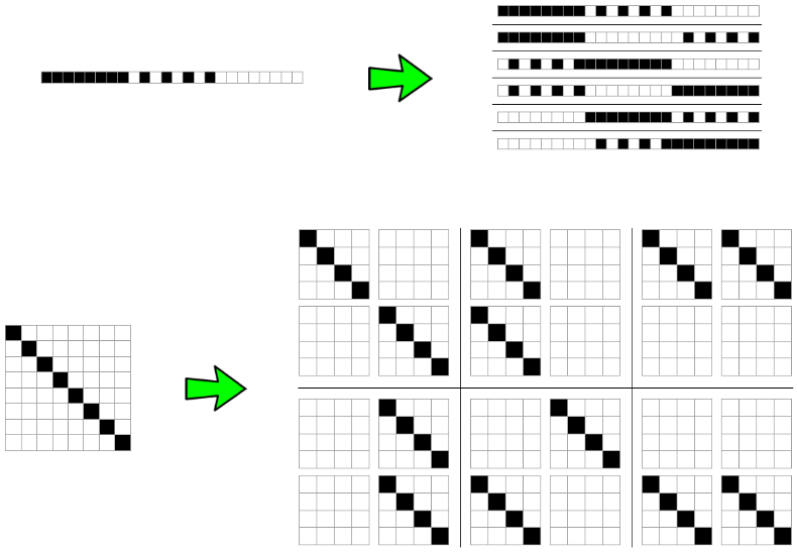


Figura 3.3: Permutaciones de bloques que comparten un mismo valor de BDM. Tomada con permiso de [ZKT23].

Los cálculos de BDM sin traslape son invariantes ante permutaciones de bloques, aún cuando estas permutaciones pueden tener diferentes complejidades debido a que las reorganizaciones de bloques pueden producir patrones estadísticos o algorítmicos. Por lo tanto, el no traslape incrementa la inexactitud del BDM en objetos aleatorios, aunque no lo haga con objetos de baja complejidad.

La partición correlacionada resuelve este problema particular de permutaciones al disminuir el número de permutaciones posibles, y así evitar la asignación inexacta del mismo valor BDM a muchas permutaciones con diferente complejidad (vea Fig. 3.3).

Ejercicio 3.1 Entre al sitio web del Calculador Algorítmico en Línea (`https://complexity-calculator.com/`) e ingrese la cadena binaria del Ejemplo 3.1 para estimar su CA. Asegúrese de introducir los mismos valores de tamaño de bloque y traslape de bloque para obtener el mismo resultado. ■

Ejercicio 3.2 Entre al sitio web del Calculador Algorítmico en Línea e ingrese cada una de las cadenas binarias del Ejercicio 2.2 para estimar su CA (nótese que, al tener estas cadenas una longitud igual a 12, se puede estimar su CA sin bloques ni traslapes, o sea, mediante CTM).

Ahora, obtenga las razones CA(`s_b`)/CA(`s_a`) y CR(`s_b`)/CR(`s_a`) (donde CR es la razón de compresibilidad, Ejercicio 2.2). ¿Cuál de las dos cadenas es más "comprimible"? ¿La respuesta depende de la métrica usada para comparar?

Por último, obtenga las razones CA(`s_c`)/CA(`s_TM`), CR(`s_c`)/CR(`s_TM`) y HBmax(`s_c`)/HBmax(`s_TM`) (donde HBmax es la entropía de bloque máxima, Ejercicio 2.1). ¿Cuál de las dos cadenas es más "comprimible"? ¿La respuesta depende de la métrica usada para comparar? Si es así, ¿a qué cree que se deba? ■

Ejercicio 3.3 Si tenemos una secuencia de longitud L y queremos implementar la partición correlacionada con bloques de tamaño l, ¿cuál es la condición que debe cumplir el parámetro de deslizamiento m en términos de L y l para tener una partición sin sobras?
¿Cuáles son los valores permitidos de m para $L = 36$ y $l = 12$?
(*Respuesta parcial*: Los tres valores permitidos más grandes de m son 6, 8 y 12). ■

3.3.2 Partición recursiva

Una alternativa a la partición correlacionada se basa en una estrategia de minimizar particiones y maximizar el tamaño de bloque u objeto base. El propósito de esta estrategia es superar estimaciones de complejidad de ajuste insuficiente o excesivo que son atribuíbles a convenciones, no sólo a limitaciones técnicas (p. ej. incomputabilidad e intratabilidad).

En la sección 3.4 se demuestra que el uso de particiones más pequeñas (es decir, particiones con un número mínimo de bloques) para BDM produce aproximaciones más precisas de la complejidad algorítmica K. Sin embargo, los costos computacionales de calcular CTM son altos. Se ha compilado una base de datos exhaustiva para matrices cuadradas de hasta 4×4. Por lo tanto, nos conviene encontrar un método para minimizar la partición de una matriz dada en cuadrados de tamaño d x $d = l$ para un determinado l. La estrategia consiste en tomar la matriz más grande múltiplo de d x d en una esquina y dividirla en submatrices cuadradas adyacentes de dicho tamaño. Luego agrupamos las celdas restantes en tres submatrices y aplicamos el mismo procedimiento, pero ahora con submatrices de tamaño $(d-1)$ x $(d-1)$. Continuamos dividiendo hasta llegar a tener submatrices de 1 x 1.

Definición 3.3.1 — Partición recursiva. Sea X una matriz de tamaño m x n con $m,n \geq d$. Denotemos por quad $= \{\text{UL}, \text{DL}, \text{UR}, \text{DR}\}$ (U = *up*, D = *down*, L = *left*, R = *right*) el conjunto de cuadrantes en una matriz, y sea quad^d el conjunto de vectores de cuadrantes de dimensión d. Definimos una función part(X,d,q_i), donde $\langle q_1, \cdots, q_d \rangle \in \text{quad}^d$, como sigue:

$$\begin{aligned} \text{part}(X,d,q_i) = &\max(X,d,q_i) \\ &\cup \text{part}(\text{resL}(X,d,q_i), d-1, q_{i+1}) \\ &\cup \text{part}(\text{resR}(X,d,q_i), d-1, q_{i+1}) \\ &\cup \text{part}(\text{resLR}(X,d,q_i), d-1, q_{i+1}) \end{aligned} \tag{3.6}$$

donde $\max(X,d,q_i)$ es el conjunto más grande de submatrices adyacentes de tamaño d x d que se pueden agrupar en la esquina que corresponde al cuadrante q_i, $\text{resR}(X,d,q_i)$ es la submatriz compuesta de todas las celdas adyacentes a la derecha que no cupieron en $\max(X,d,q_i)$ y que no

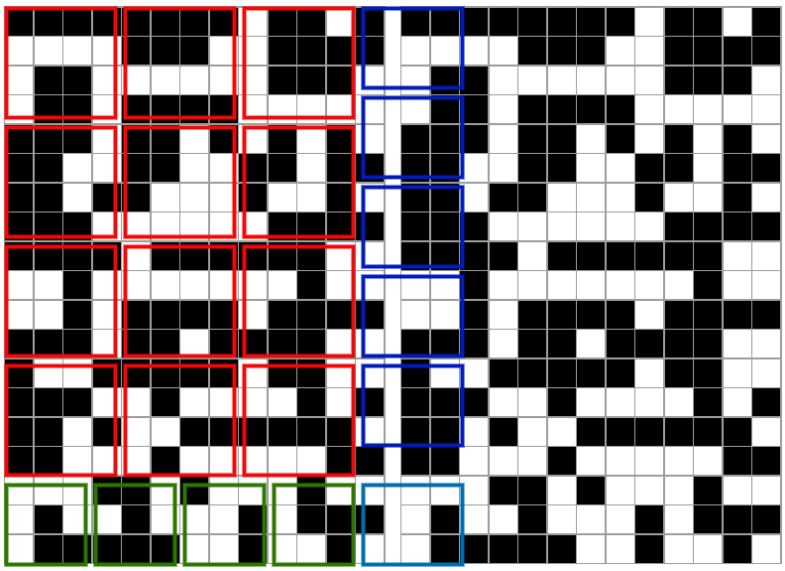


Figura 3.4: Partición recursiva de un arreglo binario bidimensional. Se comienza con bloques de tamaño 4×4 (color rojo), lo que resulta en la aparición de sobras a la derecha (R), abajo (D), y en la esquina inferior derecha (RD). En la siguiente iteración se realiza una partición con bloques de tamaño 3×3 para cada una de las tres regiones R, D y RD (colores azul, verde y cyan, respectivamente), repitiendo el arreglo de ser necesario (en este caso solo es necessario para las regiones R y RD). Se continúan las iteraciones hasta tener bloques de tamaño 1×1, o hasta que no queden sobras (nótese que solo será necesario aplicar la tercera iteración a la región R).

son parte de las celdas a la izquierda, $\text{resL}(X,d,q_i)$ es un análogo para las celdas a la izquierda y $\text{resLR}(X,d,q_i)$ es la submatriz compuesta por las celdas que pertenecen a las celdas izquierdas y derechas. Llamamos "matrices residuales" a las tres últimas submatrices.

(Esta definición se introdujo con fines de formalidad y precisión, para una visualización más fácil de interpretar, vea la Fig. 3.4). Una manera de evitar matrices residuales con diferentes tamaños de submatrices es incrustar la matriz en un toro topológico de tal manera que no se tengan fronteras en el objeto (vea Fig. 3.5). Este procedimiento, sin embargo, sobreestimará los valores de complejidad de todos los objetos (de manera desigual a lo largo de los espectros de complejidad), pero aun así permanecerá acotado.

Ejercicio 3.4 Termine la partición recursiva de la Fig. 3.4 y haga un boceto con los bloques resultantes (nótese que sólo es necesario continuar en la región R con bloques de tamaño 2×2). ■

3.4 ¿Por qué funciona el BDM? (opcional)

En esta sección se demuestran las proposiciones que nos permiten establecer al BDM como una medida híbrida entre la entropía de Shannon y la complejidad algorítmica[1]. Su lectura puede ser omitida dado que no es indispensable para la parte II de este libro.

[1] Las demostraciones están adaptadas de [ZKT23].

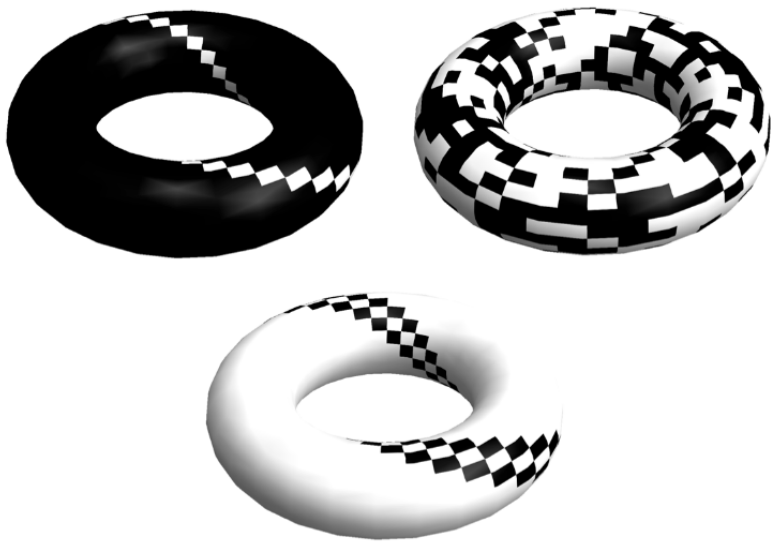


Figura 3.5: Una forma de lidiar con la descomposición de arreglos bidimensiones es incrustarlos en un toro de 2 dimensiones, haciendo que los bordes sean cíclicos o periódicos al unir los bordes del objeto. Tomada con permiso de [Zen+18].

3.4.1 Cotas superior e inferior del BDM

Proposición 3.4.1 Sea BDM la función definida en la sección 3.2, ec. (3.4), y sea X un arreglo de dimensión w. Entonces,

$$K(X) \leq \text{BDM}(X,\{x_i\}) + O(\log|A|) + \varepsilon \tag{3.7}$$

$$\text{BDM}(X,\{x_i\}) \leq \left|\text{Adj}(X)_{\{x_i\}}\right| K(X) + O\left(\left|\text{Adj}(X)_{\{x_i\}}\right| \log\left|\text{Adj}(X)_{\{x_i\}}\right|\right) - \varepsilon \tag{3.8}$$

donde A es un conjunto compuesto de todas las posibles maneras de acomodar los elementos de $\text{Adj}(X)_{\{x_i\}}$ en un arreglo de dimensión w, y ε es la suma de los errores para la aproximación CTM sobre todos los sub-arreglos usados.

Demostración:

Sea $\text{Adj}(X)_{\{x_i\}} = \{(r_j,n_j)\}_1^k$, y $\{p_j\}_1^k$, $\{t_j\}_1^k$ los programas para la MTs universales tales que

$$U(p_j) = r_j \qquad U(t_j) = n_j \tag{3.9}$$

$$K(r_j) = |p_j| \qquad |t_j| \leq 2\log n_j + c \tag{3.10}$$

Sea ε_j una constante positiva tal que $\text{CTM}(r_j) + \varepsilon_j = K(r_j)$, y sea $\varepsilon = \sum_{j=1}^k \varepsilon_j$.

Para la primera desigualdad, podemos construir un programa q_w -cuya descripción solo depende de w- tal que, dada una descripción de $\text{Adj}(X)_{\{x_i\}}$ y un índice l, enumera todas las maneras de acomodar los elementos en $\text{Adj}(X)_{\{x_i\}}$ y nos da como salida el arreglo con posición l.

Es decir, q_w construye al conjunto A y da como resultado el elemento (arreglo) con índice l.

Nótese que $|l|$ y $\left|\text{Adj}\,(X)_{\{x_i\}}\right|$ son del orden de $\log|A|$. Por lo tanto,

$$U\left(q_w, \{p_j, t_j\}_1^k, l\right) = X \tag{3.11}$$

y

$$\begin{aligned}
K(X) &\leq \left|q_w, \{p_j, t_j\}_1^k, l\right| \\
&= |q_w| + \sum_{j=1}^{k} \left(\left|p_j\right| + \left|t_j\right|\right) + |l| \\
&\leq |q_w| + \sum_{j=1}^{k} [K(r_j) + \log n_j] + \\
&+ \sum_{j=1}^{k} [c] + |l| \\
&= |q_w| + \sum_{j=1}^{k} [\text{CTM}(r_j) + \varepsilon_j + \log n_j] + \\
&+ \sum_{j=1}^{k} [c] + |l| \\
&\leq \text{BDM}(X, \{x_i\}) + \varepsilon + |q_w| + \\
&+ \left|\text{Adj}\,(X)_{\{x_i\}}\right| c + O(\log|A|) \\
&\leq \text{BDM}(X, \{x_i\}) + \varepsilon + \\
&+ O(\log|A|)\,[c+1] + |q_w| \\
&\leq \text{BDM}(X, \{x_i\}) + O(\log|A|) + \varepsilon
\end{aligned} \tag{3.12}$$

Para la segunda desigualdad, sea q_X el programa más pequeño que genera a X. Podemos describir un programa $q_{\{x_i\}}$ el cual, dada una descripción de X y un índice j, construye el conjunto $\text{Adj}\,(X)_{\{x_i\}}$ y da como salida r_j, es decir,

$$U\left(q_{\{x_i\}} q_X j\right) = r_j \tag{3.13}$$

Nótese que cada $|j|$ es del orden de $\log\left|\text{Adj}\,(X)_{\{x_i\}}\right|$. Por lo tanto,

$$\left|p_j\right| = \text{CTM}(r_j) + \varepsilon_j = K(r_j) \leq \left|q_{\{x_i\}}\right| + |q_X| + O\left(\log\left|\text{Adj}\,(X)_{\{x_i\}}\right|\right) \tag{3.14}$$

y

$$\text{CTM}(r_j) + \varepsilon_j + \log n_j \leq \left|q_{\{x_i\}}\right| + |q_X| + O\left(\log\left|\text{Adj}\,(X)_{\{x_i\}}\right|\right) + \log n_j \tag{3.15}$$

Finalmente, al sumar sobre j -nótese que los n_j son del orden de $\log|A|$,

$$\begin{aligned}
\text{BDM}(X, \{x_i\}) + \varepsilon &\leq \left|\text{Adj}\,(X)_{\{x_i\}}\right| \left[|q_X| + \left|q_{\{x_i\}}\right| + O\left(\log\left|\text{Adj}\,(X)_{\{x_i\}}\right|\right)\right] + \sum_{j=1}^{k} \log n_j \\
\text{BDM}(X, \{x_i\}) + \varepsilon &\leq \left|\text{Adj}\,(X)_{\{x_i\}}\right| \left[|q_X| + O\left(\log\left|\text{Adj}\,(X)_{\{x_i\}}\right|\right) + \left|q_{\{x_i\}}\right| + O\left(\log^2|A|\right)\right] \\
&\leq \left|\text{Adj}\,(X)_{\{x_i\}}\right| K(X) + O\left(\left|\text{Adj}\,(X)_{\{x_i\}}\right| \log\left|\text{Adj}\,(X)_{\{x_i\}}\right|\right)
\end{aligned} \tag{3.16}$$

■

Corolario 3.4.2 Si la partición definida por x_i es pequeña, esto es, si $\left|\text{Adj}(X)_{\{x_i\}}\right|$ se aproxima a 1, entonces $\text{BDM}(X,\{x_i\}) \simeq K(X)$

Demostración:

Dadas las desigualdades presentes en la Proposición 3.4.1, tenemos que

$$K(X) - O(\log|A|) - \varepsilon \leq \text{BDM}(X,\{x_i\}) \tag{3.17}$$

y

$$\text{BDM}(X,\{x_i\}) \leq \left|\text{Adj}(X)_{\{x_i\}}\right| K(X) + O\left(\left|\text{Adj}(X)_{\{x_i\}}\right| \log \left|\text{Adj}(X)_{\{x_i\}}\right|\right) - \varepsilon \tag{3.18}$$

lo que en el límite da lugar a $K(X) - \varepsilon \leq \text{BDM}(X,\{x_i\}) \leq K(X) - \varepsilon$ y $\text{BDM}(X,\{x_i\}) = K(X) - \varepsilon$. De [Sol+14] podemos decir que el error ε es pequeño, y que por el teorema de invarianza va a converger a un valor constante. ■

3.4.2 Convergencia del BDM hacia la entropía de Shannon en el peor de los casos

Sea $\{x_i\}$ una partición de X definida tal y como en las secciones previas para un d fijo. Entonces la entropía de Shannon de X para la partición $\{x_i\}$ está dada por:

$$\text{H}_{\{x_i\}}(X) = -\sum_{(r_j,n_j)\in \text{Adj}(X)_{\{x_i\}}} \frac{n_j}{|\{x_i\}|} \log\left(\frac{n_j}{|\{x_i\}|}\right) \tag{3.19}$$

donde $\text{P}(r_j) = \frac{n_j}{|\{x_i\}|}$ y el arreglo r_j se toma como un símbolo en sí mismo. La siguiente proposición establece la relación asintótica entre $\text{H}_{\{x_i\}}(X)$ y BDM.

Proposición 3.4.3 Sea M una matriz bidimensional y sea $\{x_i\}$ una estrategia de partición con elementos de tamaño máximo $d \times d$. Entonces,

$$\left|\text{BDM}_{\{x_i\}}(X) - \text{H}_{\{x_i\}}(X)\right| \leq O(\log|\{x_i\}|) \tag{3.20}$$

Demostración:

Dado que, tanto el conjunto de matrices de tamaño $d \times d$ como el valor máximo para $\text{CTM}(r_j)$ son finitos, existe una constante c_d tal que $\left|\text{Adj}(X)_{\{x_i\}}\right| \text{CTM}(r_j) \leq c_d$ ó $\left|\text{Adj}(X)_{\{x_i\}}\right| \text{máx}\left\{\text{CTM}(r_j)\right\} = c_d$. Por lo tanto:

$$\begin{aligned}
\text{BDM}_{\{x_i\}}(X) - \text{H}_{\{x_i\}}(X) &= \sum_j \left(\text{CTM}(r_j) + \log(n_j) + \frac{n_j}{|\{x_i\}|}\log\left(\frac{n_j}{|\{x_i\}|}\right)\right) \\
&\leq \sum_j \left(\frac{c_d}{\left|\text{Adj}(X)_{\{x_i\}}\right|} + \log(n_j) + \frac{n_j}{|\{x_i\}|}\log\left(\frac{n_j}{|\{x_i\}|}\right)\right) \\
&= c_d + \sum_j \left(\log(n_j) + \frac{n_j}{|\{x_i\}|}\log\frac{n_j}{|\{x_i\}|}\right)
\end{aligned} \tag{3.21}$$

también nótese que $\sum_j n_j = |\{x_i\}|$. Por lo tanto existe c'_d tal que $\frac{n_j}{|\{x_i\}|} \leq c'_d$ y

$$\begin{aligned}
\mathrm{BDM}_{\{x_i\}}(X) - \mathrm{H}_{\{x_i\}}(X) &\leq c_d + \sum_j \left(\log\left(c'_d\,|\{x_i\}|\right) + c'_d \log\left(c'_d\right)\right) \\
&= c_d + \left|\mathrm{Adj}(X)_{\{x_i\}}\right| \left[\left(c'_d + 1\right)\log\left(c'_d\right) + \log|\{x_i\}|\right] \\
&= c_d \left[1 + \frac{1}{\max\left\{\mathrm{CTM}(r_j)\right\}}\left[\left(c'_d + 1\right)\log\left(c'_d\right) + \log|\{x_i\}|\right]\right] \\
&= c_d\, O(\log|\{x_i\}|) \qquad (3.22)
\end{aligned}$$

■

Ahora bien, es importante señalar que la prueba anterior establece el límite en términos de la constante c_d, cuyo valor se define en términos de matrices para las cuales se ha calculado el valor CTM, i.e., $c_d = \left|\mathrm{Adj}(X)_{\{x_i\}}\right| \text{máx}\left\{\mathrm{CTM}(r_j)\right\}$. Por lo tanto,

- En el peor de los casos (c_d pequeño), es decir, cuando el CTM se ha calculado para un número comparativamente pequeño de matrices, o la matriz base más grande tiene una complejidad algorítmica baja, el comportamiento de BDM es similar a la entropía.
- En el mejor de los casos (c_d grande), cuando el CTM se actualiza por cualquier medio, el BDM se aproxima a la complejidad algorítmica.

En resumen, el Método de Descomposición en Bloques (BDM) nos permite aproximar la complejidad algorítmica de sistemas complejos de una manera más manejable. Al descomponer un sistema en piezas más pequeñas y calcular la complejidad de cada pieza, podemos obtener una estimación de la complejidad total del sistema. Esta técnica se basa en la teoría de la información de Shannon y proporciona una medida intermedia entre la entropía y la complejidad algorítmica, lo que nos ayuda a entender mejor la estructura y el comportamiento de sistemas complejos.

II Parte II: Implementación y Aplicaciones

4 Secuencias binarias 31
4.1 Introducción a la librería PyBDM
4.2 Cadenas binarias aleatorias
4.3 Secuencia Thue-Morse
4.4 Conclusiones

5 Arreglos bidimensionales y Grafos 41
5.1 Grafo ZK
5.2 Conclusiones

Bibliografía 49

4. Secuencias binarias

En el Capítulo 3 vimos que se puede usar el Calculador Algorítmico en Linea para estimar la CA de cadenas binarias por BDM, siendo posible controlar el tamaño de bloque y el traslape entre bloques. En este capítulo veremos cómo implementar la librería PyBDM[Tal24] para realizar cálculos BDM de forma más intensiva, con un volumen de datos mucho mayor y un control sobre la forma en que podemos presentar nuestros resultados. Llevaremos a cabo cálculos BDM sobre algunos tipos de secuencias binarias unidimensionales, y con los resultados podremos confirmar la primacía que tiene la CA a la hora de estimar la aleatoriedad en los objetos y revelar la existencia de patrones algorítmicos subyacentes.

4.1 Introducción a la librería PyBDM

PyBDM es una librería Python que implementa BDM mediante el enfoque orientado a objetos y recibe entradas representadas como arreglos Numpy de tipo entero. Su instalación es por medio de la instrucción `pip install pybdm` en la terminal.

En esta introducción vamos a implementar BDM en la cadena binaria del Ejemplo 3.1 como ilustración. Empecemos importando las librerías necesarias[1],

```
import numpy as np
from pybdm import BDM

s = "0101010101010101" # La cadena del Ejemplo 3.1
s_array = np.array(list(s), dtype=int) # Convertimos a arreglo
```

Nótese que se está importando la clase `BDM`. A continuación, inicialicemos un objeto de dicha clase, y computemos BDM en nuestro arreglo mediante el método `bdm`,

```
# El argumento ndim especifica la dimensionalidad del objeto BMD
bdm = BDM(ndim=1)
```

[1] En este trabajo omitimos el uso de acentos y caracteres fuera del alfebeto inglés (tales como la "ñ") en los comentarios a los listados de código.

```
# Computar BDM
bdm_value = bdm.bdm(s_array)
# Tambien se puede computar la entropia de Shannon base 2
entropy_value = bdm.ent(s_array)

print(round(bdm_value, 2), round(entropy_value, 2))
```

Los valores BDM y entropía son 26.99 y 0.0, respectivamente.

En el último listado de código, el objeto inicializado `bdm` es tal que, por default, implementa particiones ignorando las sobras en los objetos. A continuación, el siguiente listado muestra cómo se inicializan objetos de la clase `BDM` que implementan los diferentes tipos de partición (vea Secc. 3.3),

```
import numpy as np
from pybdm import BDM
# Importamos las diferentes clases de particion
from pybdm import PartitionIgnore, PartitionRecursive, PartitionCorrelated

s = "0101010101010101" # La cadena del Ejemplo 3.1
s_array = np.array(list(s), dtype=int) # Convertimos a arreglo

# Default
bdm_ignore = BDM(ndim=1, partition=PartitionIgnore)

#Particion recursiva
#El argumento min_length especifica el tamano minimo
#de bloque a alcanzar en la particion
bdm_recursive = BDM(ndim=1, partition=PartitionRecursive, min_length=2)

#Particion correlacionada
bdm_correlated = BDM(ndim=1, partition=PartitionCorrelated)
# Tamano de deslizamiento m igual a 1 por default,
# equivalente a
#bdm_correlated = BDM(ndim=1, partition=PartitionCorrelated, shift=1)

bdm_value = bdm_correlated.bdm(s_array)
print(round(bdm_value,3))
```

El valor BDM que se obtiene por partición correlacionada para nuestra secuencia binaria es de 57.566, al igual que en el Ejemplo 3.1.

Para mayores detalles sobre cómo usar la librería PyBDM, puede consultar la documentación disponible en el sitio web `https://pybdm-docs.readthedocs.io/en/latest/`.

Ejercicio 4.1 Tenemos la sucuencia `s = "11111111111101010101010000000000000"`, que se forma al concatenar las cadenas `s_a = 111111111111`, `s_b = 010101010101` y `s_0 = 000000000000` en ese orden. Elabore un script en Python que haga lo siguiente:

1. Calcular el BDM de todas las $3! = 6$ permutaciones posibles, con la partición `IgnoreRemainders`. Confirme que, para esta partición, el BDM es invariante ante dichas permutaciones.
2. Hacer lo mismo que en el punto anterior, pero ahora con la partición `Correlated` y para cada uno de los valores permitidos de parámetro de deslizamiento m (vea Ejercicio 3.3). También calcule la desviación estandar relativa de los valores BDM obtenidos en función del parámetro m.

¿Qué sucede con los valores BDM calculados al disminuir m? ¿Y los valores calculados de desviación estándar relativa? ¿Son significativos? ■

Ejercicio 4.2 Repetir el Ejercicio 4.1, pero ahora con tres cadenas binarias aleatorias de tamaño 12. ■

4.2 Cadenas binarias aleatorias

En esta sección analizamos conjuntos de cadenas binarias aleatorias, donde el número de 1's tiene una distribución de probabilidad binomial. Para entender las cadenas binarias aleatorias, imaginemos un experimento simple en el que lanzamos una moneda varias veces y registramos los resultados como una secuencia de 0's y 1's, donde 0 representa "cara" y 1 representa "cruz". Por ejemplo, si lanzamos la moneda tres veces y obtenemos "cara", "cruz" y "cara", registraríamos la secuencia "010". En este caso, la moneda tiene una probabilidad igual de caer en "cara" o "cruz", por lo que cada secuencia es igualmente probable. Sin embargo, si la moneda estuviera cargada, algunas secuencias podrían ser más probables que otras. En el estudio de las cadenas binarias aleatorias nos interesa analizar la distribución de estas secuencias y medir su aleatoriedad utilizando diferentes métodos, como la entropía o la complejidad algorítmica.

El procedimiento a seguir es:

1. Fijar la longitud l de las secuencias.
2. Crear una lista de secuencias $\{s_i\}_{i=0}^{100}$, donde cada s_i sea una secuencia de l dígitos binarios que resulte de l "lanzamientos de moneda" con probabilidad de "águila" (1) igual a $p = i/100$.
3. Calcular las medidas de aleatoriedad $\{A(s_i)\}_{i=0}^{100}$ (las medidas A de aleatoriedad serán la entropía, la compresibilidad por BZip2 y el BDM).
4. Hacer un gráfico donde se muestren las medidas de aleatoriedad.

En el siguiente listado se muestra la implementación de dicho procedimiento -los cálculos BDM se llevan a cabo mediante partición `IgnoreRemainders`,

```
from math import log2
import random as rd
import numpy as np
from pybdm import BDM
from matplotlib import pyplot as plt
import bz2 # Libreria BZip2


def entropy(array):
    """Esta funcion calcula la entropia
    de un arreglo binario"""
    H = 0.0
    p = np.count_nonzero(array) / len(array)
    if p != 0.0 and p != 1.0:
        H -= (p*log2(p) + (1.0-p)*log2(1.0 - p))
    return H


def biased_coin(p):
    """Esta funcion simula una moneda sesgada
    con probabilidad p de caer en aguila (1)"""
    rd_x = rd.random()
    if 0 <= rd_x and rd_x <= p:
        return 1
    else:
```

```
        return 0


def bernoulli_essay(str_len, p):
    """Esta funcion devuelve una simulacion
    de str_len lanzamientos con una p-moneda
    sesgada, en formato de arreglo"""
    return np.array([biased_coin(p) for i in range(str_len)])


def arr_to_string(arr):
    """Esta funcion convierte un arreglo a
    cadena de caracteres"""
    a_str = ""
    for nb in arr:
        a_str += str(nb)
    return a_str

bdm = BDM(ndim=1)

#PASO 1
str_len = 10000

#PASO 2
nb_arrays = 101
bin_arrs = np.array([bernoulli_essay(str_len, f/100)
                     for f in range(nb_arrays)])

#PASO 3
arrs_bdm = np.array([bdm.bdm(arr) for arr in bin_arrs])
arrs_H = np.array([entropy(arr) for arr in bin_arrs])

arrs_zip = np.zeros(nb_arrays, float)
for i in range(nb_arrays):
    a_string = bytes(arr_to_string(bin_arrs[i]), 'ascii')
    comp_string = bz2.compress(a_string)
    arrs_zip[i] = len(comp_string)
#normalizacion
arrs_zip /= max(arrs_zip)
arrs_bdm /= max(arrs_bdm)

#PASO 4
plt.plot(range(nb_arrays), arrs_H, "-o", label="Entropia")
plt.plot(range(nb_arrays), arrs_zip, "-o", label="BZip2")
plt.plot(range(nb_arrays), arrs_bdm, "-o", label="BDM")
plt.title(f"Longitud de secuencia = {str_len}")
plt.xlabel("Probabilidad p (%)")
plt.ylabel("Aleatoriedad")
plt.legend()
plt.show()
```

Al ejecutar este programa con diferentes asignaciones de `str_len` se obtienen las gráficas de la Fig. 4.1, en donde se observa una convergencia en las curvas de aleatoriedad al aumentar la longitud de secuencia. Para longitud de secuencia igual a 10000 se observan curvas bien definidas de aleatoriedad, y se pueden destacar las siguientes observaciones:

- La curva de entropía es simétrica y se ajusta a la curva de entropía para una distribución de probabilidad binaria.
- La curva de compresibilidad por BZip2 se superpone a la curva de entropía. Esto nos indica

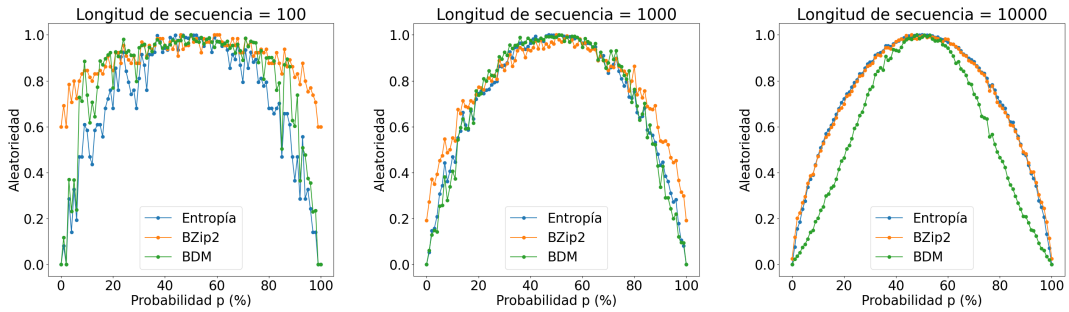


Figura 4.1: Medidas de aleatoriedad para secuencias binarias aleatorias en función de la probabilidad p. Cada gráfico corresponde a una longitud de secuencia determinada.

que el algoritmo de compresión es incapaz de revelar patrones no estadísticos en las cadenas aleatorias más allá de lo que nos puede revelar la métrica de entropía.

- La complejidad algorítmica es menor o igual que la entropía en todo el dominio de valores de p.

De hecho, entre las curvas de entropía y BDM existen "brechas de causalidad", es decir, que las respectivas cadenas binarias tienen menor aleatoriedad que la asignada por la métrica de entropía, y por lo tanto tienen más de un mecanismo generador, no obstante su valor medio o alto en entropía.

Ejercicio 4.3 Repetir el procedimiento realizado en esta sección, pero ahora con cadenas binarias alternadas. Para ello, modifique el paso 2 como sigue:

- Crear una lista de secuencias $\{s_i\}_{i=0}^{100}$, donde cada s_i sea una secuencia de l dígitos binarios alternados con una cantidad de 1's igual a $\frac{i}{100}l$.

Para crear una secuencia de l dígitos binarios alternados con cierta cantidad m de 1's, se empieza creando una secuencia de l ceros. Luego, se insertan 1's en las posiciones de la secuencia inicial con índice par en orden creciente. Si tales posiciones se agotan, se insertan los 1's restantes en las posiciones de la secuencia inicial con índice impar en orden creciente. Ejemplos:

- $l = 10, m = 2$, 0000000000 ->0101000000
- $l = 10, m = 5$, 0000000000 ->0101010101
- $l = 10, m = 7$, 0000000000 ->1111010101

Una vez obtenidas las correspondientes gráficas, responda las siguientes preguntas:

- ¿Cambian las gráficas de entropía al aumentar la longitud de secuencia? ¿A qué se debe esto?
- ¿Nota algún patrón en las curvas de BDM? ¿A qué cree que se deba?
- ¿El BDM es menor que la entropía en todo el dominio de porcentajes? Si no es así, cómo lo puede explicar?
- ¿El BDM sigue siendo más efectivo que la compresibilidad (por BZip2) para comprimir las cadenas alternadas?

(Opcional: Repetir el ejercicio implementando las otras dos particiones). ■

Figura 4.2: Secuencia de Thue-Morse y cómo se obtiene.

4.3 Secuencia Thue-Morse

Para terminar este capítulo, vamos a analizar una secuencia muy peculiar con las herramientas del BDM, lo que a su vez nos brindará un ejemplo muy ilustrativo de que la (aparente) ausencia de patrones estadísticos no implica la ausencia de otros patrones, en específico patrones algorítmicos.

En algunos ejercicios de los Capítulos 2 y 3 estuvo presente la secuencia `s_TM = "011010011001"`, la cual parece ser una secuencia aleatoria más, pero que tiene una compresibilidad mayor a la de la cadena `s_c = "111001011000"`, si se mide desde la complejidad algorítmica, mientras que, si se mide desde la razón de compresibilidad sin pérdidas, ambas cadenas son indistinguibles (vea Ejercicio 3.2). Esto no es casualidad, como veremos en un momento.

En matemáticas, la sucesión de Thue-Morse[AS03] es una sucesión de dígitos binarios que si se concatenan producen una secuencia con segmentos iniciales alternos. La secuencia se obtiene comenzando con un cero y añadiendo sucesivamente el complemento Booleano de la secuencia que existe al momento (vea Fig. 4.2).

Ejercicio 4.4 En el listado que sigue a continuación en el texto, se encuentra el siguiente bloque de código Python,

```
def binary_complement(b_arr):
    return [(digit+1)%2 for digit in b_arr]

def thue_morse_seq(n):
```

donde la función `binary_complement(b_arr)` recibe una lista de dígitos binarios y devuelve una lista con los dígitos del complemento Booleano.

Complete la definición de la función `thue_morse_seq(n)`, donde `n` es el nivel de la secuencia que se desea generar, siendo `n = 1` el nivel más interno o nivel más inferior, vea la Fig. 4.2. Implemente su código y verifique que los dígitos obtenidos sí forman parte de la secuencia Thue-Morse.

(*Sugerencia*: Será más fácil si lo hace de forma recursiva). ■

Para analizar esta secuencia, llevemos a cabo el siguiente procedimiento:

1. Generar cierta cantidad de dígitos de la secuencia TM (en este caso, realizamos 9 iteraciones para generar $2^9 = 512$ dígitos).
2. Crear un arreglo inicial vacío
3. Tomar los primeros 12 dígitos de la secuencia TM generada y pasarlos a la lista inicial
4. Calcular la entropía, la entropía de bloque y el BDM de la lista inicial.
5. Repetir los pasos 3 y 4 hasta que la secuencia TM quede vacía. Así tendremos medidas de aleatoriedad en función del número de bloques tomados de la secuencia TM.
6. Graficar el resultado.

A continuación se muestra dicho procedimiento implementado con un código de Python -al igual que

en la sección anterior, los cálculos BDM se llevan a cabo mediante partición `IgnoreRemainders`,

```
import numpy as np
from pybdm import BDM
from pybdm import PartitionIgnore, PartitionRecursive, PartitionCorrelated
from math import log2
from matplotlib import pyplot as plt


def binary_complement(b_arr):
    return [(digit+1)%2 for digit in b_arr]

#Ejercicio 4.4
def thue_morse_seq(n):
    #Complete la definicion

#Ejercicio 2.1
def entropy(p_dist):
    I = 0.0
    for p in p_dist:
        if p != 0:
            I += (-p*log2(p))
    return I

def block_entropy(a_string, block_length=1):
    #Complete la definicion


def nb_array_to_string(array):
    """Esta funcion convierte un arreglo binario
      a cadena de caracteres"""
    a_str = ""
    for n in array:
        a_str += str(n)
    return a_str


bdm = BDM( ndim = 1)

#PASO 1
NB_ITERATIONS = 9
NB_BLOCKS = int((2**NB_ITERATIONS)/12)
tm_seq = np.array(thue_morse_seq(NB_ITERATIONS))
tm_string = nb_array_to_string(tm_seq)

#PASOS 2, 3, 4 y 5
sample_H = np.array([my_entropy(tm_string[0:(i+1)*12])
    for i in range(NB_BLOCKS)])
sample_blockH = np.array([bdm.ent(tm_seq[0:(i+1)*12], normalized=True)
    for i in range(NB_BLOCKS)])
#BDM normalizado
sample_bdm = np.array([bdm.bdm(tm_seq[0:(i+1)*12], normalized=True)
    for i in range(NB_BLOCKS)])

#PASO 6
plt.plot(range(1, NB_BLOCKS+1), sample_H, "-o", label="Entropia")
plt.plot(range(1, NB_BLOCKS+1), sample_blockH, "-o",
    label="Entropia␣de␣bloque")
plt.plot(range(1, NB_BLOCKS+1), sample_bdm, "-o", label="BDM")
plt.title("Secuencia␣de␣Thue-Morse␣(primeros␣512␣digitos)
␣␣␣␣\n␣Tamano␣de␣bloque:␣12")
```

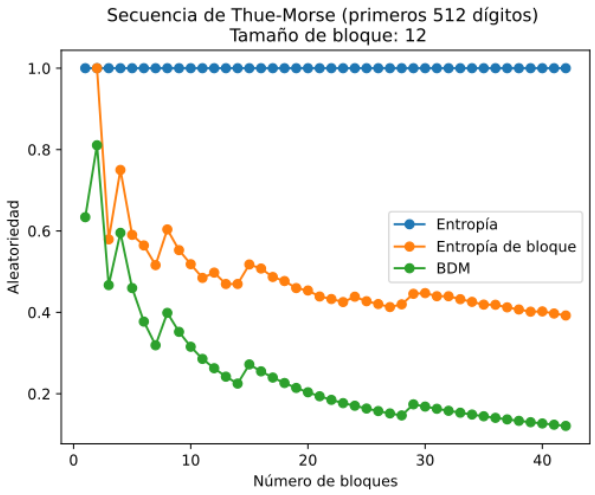


Figura 4.3: Entropía, entropía de bloque y BDM para la secuencia Thue-Morse.

```
plt.xlabel("Numero de bloques")
plt.ylabel("Aleatoriedad")
plt.legend()
plt.show()
```

Al ejecutar este script se obtiene la gráfica de la Fig. 4.3, de donde saltan a la vista un par de hechos:

- La entropía se mantiene en su valor máximo sin importar el número de bloques que se agreguen. Esto nos indica que la entropía es incapaz de detectar los patrones no estadísticos en la secuencia.
- El BDM disminuye notablemente al "alimentarlo" con más y más bloques, por lo que se puede decir que BDM es capaz de detectar patrones algorítmicos en la secuencia. La gráfica de entropía de bloque tiene un comportamiento similar, pero asigna una mayor aleatoriedad, por lo que es menos eficiente para detectar patrones no estadísticos.

Ejercicio 4.5 Repetir el Ejercicio 4.1, pero ahora con los tres primeros bloques tamaño 12 de la secuencia Thue-Morse. ¿Nota alguna diferencia respecto a aquel ejercicio y el Ejercicio 4.2? ■

4.4 Conclusiones

A partir del análisis de los resultados obtenidos en este capítulo se pueden extraer las siguientes conclusiones:

- Se confirma que el BDM es capaz de "comprimir" más objetos de manera algorítmica, es decir, que puede asignar menor aleatoriedad a más objetos que la entropía y la compresión sin pérdidas (vea Subsección 3.2.1)
- La compresibilidad es incapaz de detectar patrones más allá de lo que es capaz la entropía cuando se trata de objetos estadísticamente aleatorios, mientras que el BDM es capaz de detectar "brechas de causalidad" (vea Fig. 4.1).
- El BDM es capaz de detectar progresivamente patrones algorítmicos si se le alimenta con porciones cada vez mayores de un objeto, mientras que la entropía es incapaz o menos eficiente si el objeto carece de patrones estadísticos (vea Fig. 4.3).

Los resultados obtenidos sugieren que la CA (estimada por BDM) es la mejor métrica para evaluar

la aleatoriedad en los objetos.

5. Arreglos bidimensionales y Grafos

En este capítulo veremos cómo se pueden analizar grafos o redes mediante BDM. Para ello, vamos a analizar un tipo de grafo cuya construcción es sencilla, pero que posee algunas propiedades que lo hacen interesante. Veremos que, al poder estimar la complejidad de arreglos bidimensionales con ayuda de la librería `bdm`, es posible estimar la complejidad de una red o grafo desde su **matriz de adyacencia**. Además, con los resultados obtenidos podremos establecer conclusiones parecidas a las del Capítulo 4 sobre la CA como la métrica más óptima para comprimir y estimar la aleatoriedad.

5.1 Grafo ZK

El grafo ZK[ZKT17] es un grafo simple que nos permitirá ilustrar las limitaciones de la entropía como métrica de aleatoriedad. Se construye de la siguiente manera:

- Sea un nodo con etiqueta 1. Si un nodo con etiqueta n tiene grado n, lo llamamos un *nodo central*; de otro modo lo llamamos *nodo de soporte*.
- De forma iterativa, hacer que el nodo de soporte con la etiqueta más pequeña sea un nodo central mediante la adición de vínculos hacia nodos, potencialmente nuevos, que siguen inmediatamente.

En el siguiente listado se muestra la construcción del grafo ZK con 8 iteraciones en lenguaje Python. Su ejecución da lugar a la ilustración de la Fig. 5.1; en cada paso se pueden observar cuáles son los nodos centrales (en rosa) y cuáles son los nodos de soporte (en cyan).

```
import networkx as nx
import matplotlib.pyplot as plt

def ZK_graph(step_nb):
    G = nx.Graph()
    G.add_node(1)
    for support_node in range(1, step_nb + 1):
        nb_available_edges = support_node - G.degree[support_node]
        for i in range(1, nb_available_edges + 1):
            G.add_edge(support_node, support_node + i)
```

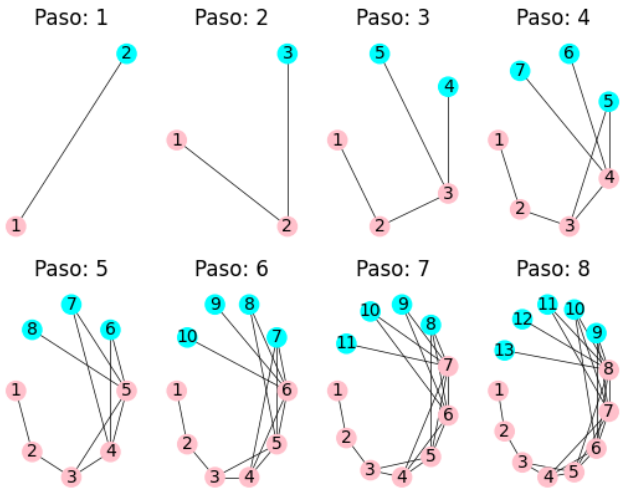


Figura 5.1: Una serie de pasos en la construcción del grafo ZK.

```
    return G

my_Gs = [ZK_graph(step_nb) for step_nb in range(1,9)]
color_maps = [[] for _ in range(8)]
# Nodos centrales en rosa, nodos de soporte en cyan
for i in range(8):
    for node in (my_Gs[i]):
        if node == (my_Gs[i]).degree[node]:
            (color_maps[i]).append('pink')
        else:
            (color_maps[i]).append('cyan')


options = [ {
    'width': 0.5,
    'font_size': 10,
    'node_size': 120,
    'with_labels': True
} for _ in range(8)]
for _ in range(8):
    options[_]['node_color'] = color_maps[_]

#Dibujamos grafos
m = 240 #2 filas y 4 columnas
for i in range (1,9):
    subax1 = plt.subplot(m+i)
    nx.draw_shell(my_Gs[i-1], **options[i-1])
    subax1.set_title(f'Step: {i}')

plt.show()
```

La secuencia de grados de los nodos etiquetados es $d = 1,2,3,...,n$ y es la constante de Champernowne en base 10, un número real trascendental cuya expansión decimal es normal de Borel. Pero la matriz de adyacencia es dispersa. Por lo tanto, del grafo ZK se podrían obtener conclusiones contradictorias según sea la característica que se analice -secuencia de grados o matriz de adyacencia

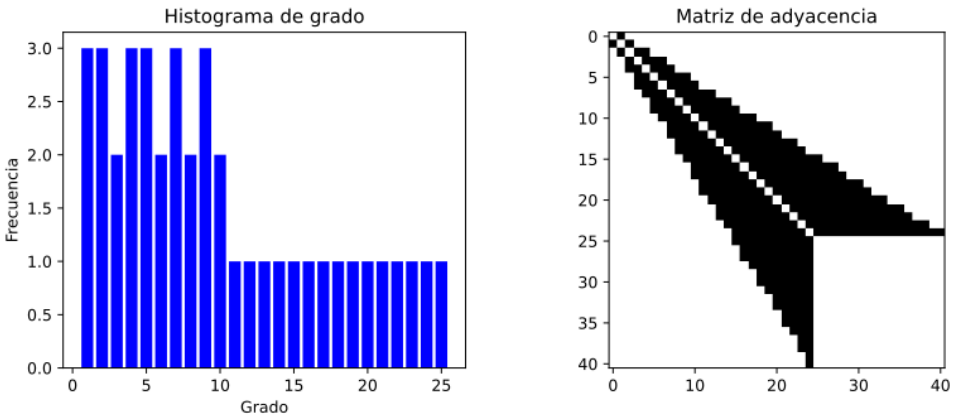


Figura 5.2: Histograma de grado y matriz de adyacencia para el grafo ZK (paso 25).

(vea Fig. 5.2). Para ver por qué, vamos a calcular la entropía para cada una de dichas características,

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import entropy


def two_rate(nb_list):
    """Esta funcion toma una lista de numeros, junta los digitos
    y devuelve una lista de numeros de dos digitos"""
    str_list = [str(n) for n in nb_list]
    string = ''.join(map(str,str_list))
    return [int(string[i:i+2]) for i in range(0, len(string), 2)]

#Numero de pasos con que se construye nuestro grafo ZK
nb_steps = 200
#Inicializamos listas de entropia
I_Adjacency = [] #Para matriz de adyacencia
I_Sequence = [] #Para secuencia de grados
I_Sequence_two_rate = [] #Para secuencia de grados (2-bloques)


G = nx.Graph()
G.add_node(1)
# Calculamos entropias en cada paso de construccion
for support_node in range(1, nb_steps + 1):
    nb_available_edges = support_node - G.degree[support_node]
    for i in range(1, nb_available_edges + 1):
        G.add_edge(support_node, support_node + i)

    # Matriz de adyacencia
    A = nx.to_numpy_array(G, dtype=int)
    # Conteo de frecuencias
    unique, frequency = np.unique(A, return_counts = True)
    # Entropia desde la matriz de adyacencia
    I_Adjacency.append(entropy(frequency))
```

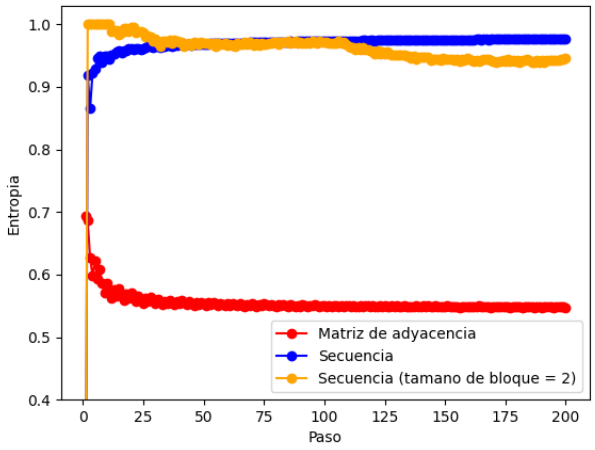


Figura 5.3: Medidas de aleatoriedad por entropía del grafo ZK en función del número de pasos tomados para generarlo. Se toman en cuenta dos características del objeto: secuencia de grados (tamaños de bloque 1 y 2) y matriz de adyacencia.

```
    #Secuencia de grados
    degree_sequence = [d for n, d in G.degree()]
    #Secuencia de grados (2-bloques)
    degree_seq_two_rate = two_rate(degree_sequence)
    # Conteo de frecuencias y entropias
    unique, frequency = np.unique(degree_sequence, return_counts = True)
    I_Sequence.append(entropy(frequency, base=len(frequency)
                              if len(frequency)>1 else 2))
    unique, frequency = np.unique(degree_seq_two_rate,
        return_counts = True)
    I_Sequence_two_rate.append(entropy(frequency, base=len(frequency)
                                       if len(frequency)>1 else 2))

#Graficamos
steps = list(range(1, nb_steps+1))
plt.plot(steps, I_Adjacency,"-o",color="red", label="Matriz de adyacencia")
plt.plot(steps, I_Sequence,"-o", color='blue', label="Secuencia")
plt.plot(steps, I_Sequence_two_rate,"-o", color='orange',
         label="Secuencia (tamano de bloque = 2)")
plt.ylim([0.4,1.03])
plt.xlabel("Paso")
plt.ylabel("Entropia")
plt.legend()
plt.show()
```

Al ejecutar este script se obtiene la gráfica de la Fig. 5.3, donde vemos que las curvas de entropía convergen a un valor fijo, pero sobre todo vemos que se pueden extraer conclusiones contradictorias sobre aleatoriedad: según la secuencia de grados, el grafo ZK es altamente aleatorio, mientras que, según la matriz de adyacencia, su entropía es media.

Por último, vamos a calcular la razón de compresibilidad (por LZW) y el BDM de nuestro grafo en función del número de pasos, a partir de la matriz de adyacencia,

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
```

```
from pybdm import BDM, PartitionIgnore
from scipy.stats import entropy

# Del Ejercicio 2.2
def compress_ratio(uncompressed):
    """Comprime una cadena , y devuelve la razon de compresibilidad."""
    initial_length = len(uncompressed)

    keys_list = list(sorted(set(uncompressed)))
    dict_size = len(keys_list)
    dictionary = {keys_list[i]:i+1 for i in range(dict_size)}

    result = []
    while len(uncompressed) > 0:
        w = uncompressed[0]
        index = 1
        while w in dictionary:
            wc = w
            if index < len(uncompressed):
               w += uncompressed[index]
            else:
                break
            index += 1
        result.append(dictionary[wc])
        uncompressed = uncompressed.replace(wc, '', 1)
        dictionary[w] = dict_size + 1
        dict_size += 1

    result_string = ''
    for n in result:
        result_string += str(n)

    return len(result_string)/initial_length

# Numero de pasos con que se construye nuestro grafo ZK
nb_steps = 250
bdm = BDM(ndim=2, partition=PartitionIgnore)
# Inicializamos listas de BDM y compresibilidad
K_Adjacency = []
compressibility = []


G = nx.Graph()
G.add_node(1)
# Calculamos en cada paso de construccion
for support_node in range(1, nb_steps + 1):
    nb_available_edges = support_node - G.degree[support_node]
    for i in range(1, nb_available_edges + 1):
        G.add_edge(support_node, support_node + i)

    # Calculamos BDM
    A = nx.to_numpy_array(G, dtype=int)
    # Con esta condicion evitamos una excepcion
    BDM_value = bdm.bdm(A, normalized=True) if support_node > 2 else 0.0
    K_Adjacency.append(BDM_value)
    # Calculamos compresibilidad
    A_flattened = np.ndarray.flatten(A, order='C')
    A_string = ""
    for digit in A_flattened:
        A_string += str(digit)
    compressibility.append(compress_ratio(A_string))
```

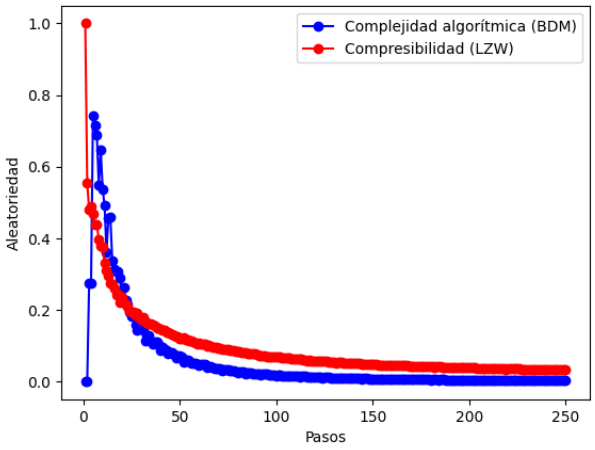


Figura 5.4: Medidas de aleatoriedad por compresibilidad LZW y BDM del grafo ZK en función del número de pasos tomados para generarlo.

```
#Graficamos
steps = range(1, nb_steps+1)
plt.plot(steps, K_Adjacency,"-o",color="blue",
    label="Complejidad␣algoritmica␣(BDM)")
plt.plot(steps, compressibility,"-o",color="red",
    label="Compresibilidad␣(LZW)")
plt.xlabel("Pasos")
plt.ylabel("Aleatoriedad")
plt.legend()
plt.show()
```

de donde se obtiene la gráfica de la Fig. 5.4. Al compararla con la gráfica de la Fig. 5.3, vemos que LZW y BDM fueron más eficientes que la(s) entropía(s) para detectar patrones no estadísticos. También se observa que la compresibilidad aproxima a la complejidad algorítmica desde arriba.

Ejercicio 5.1 Vuelva a calcular la compresibilidad LZW y el BDM para el grafo ZK, pero ahora hágalo a partir de la secuencia de grados. ■

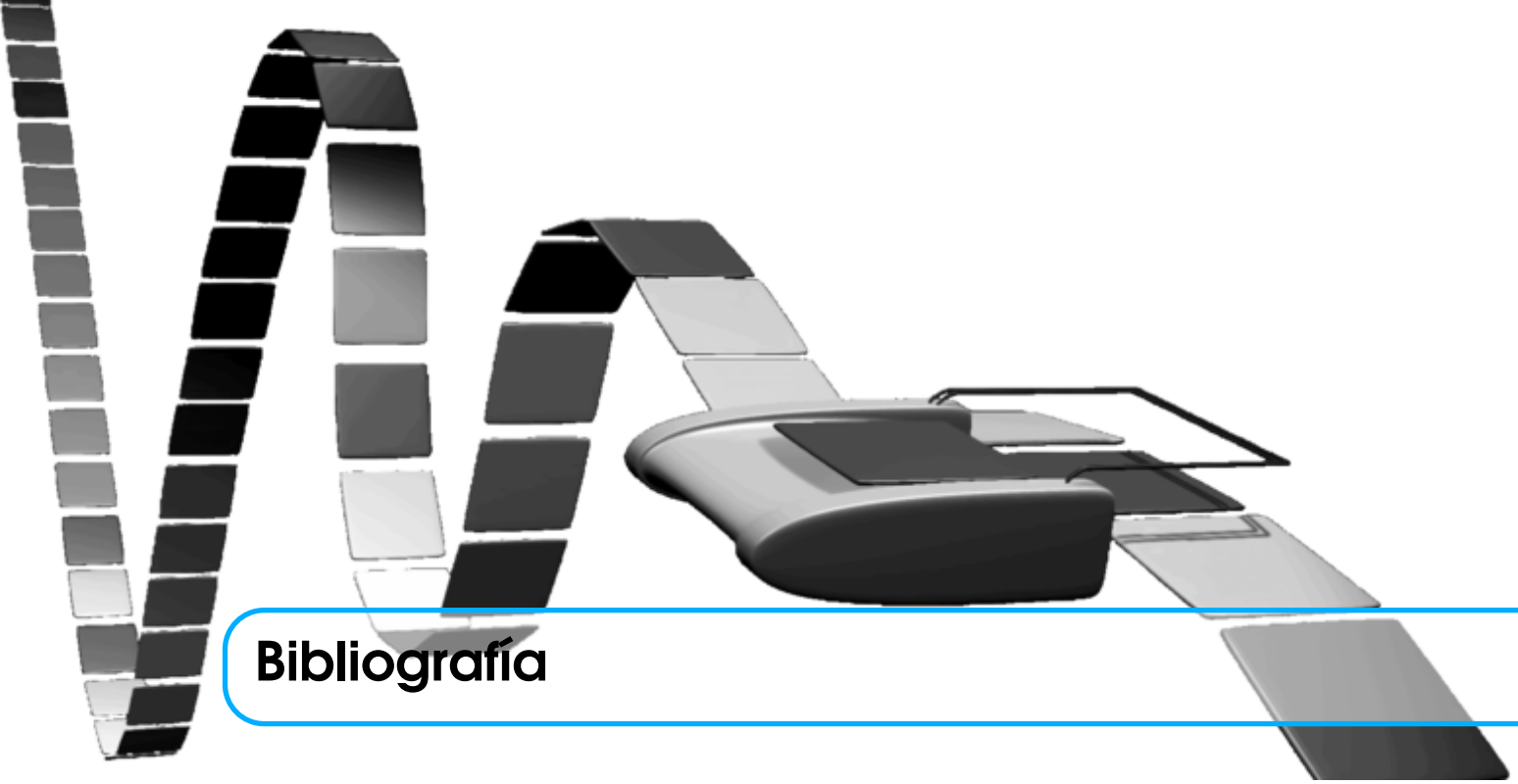
5.2 Conclusiones

A partir del análisis de los resultados obtenidos en este capítulo se pueden extraer las siguientes conclusiones:

- La entropía no es invariante ante diferentes descripciones de un objeto (vea Fig. 5.3).
- Se confirma que la compresibilidad sin pérdidas aproxima a la complejidad algorítmica, pero sólo en algunos casos, y por lo tanto la compresibilidad es una condición suficiente pero no necesaria de baja complejidad algorítmica (vea Sec. 2.3 y Figs. 4.1 y 5.4).

Este capítulo y el capítulo anterior subrayan la importancia de la Complejidad Algorítmica como una herramienta poderosa para entender la naturaleza y la estructura de los sistemas complejos. A través de los ejemplos y aplicaciones presentados, hemos demostrado que la Complejidad Algorítmica proporciona una perspectiva única y profunda que va más allá de medidas tradicionales como la entropía. Al adoptar esta metodología, podemos descubrir patrones y regularidades ocultas

en los datos, ofreciendo así una comprensión más completa y matizada de la complejidad inherente a los sistemas que estudiamos. En resumen, la Complejidad Algorítmica se consolida como un enfoque esencial para desentrañar los misterios de la complejidad en la ciencia y la tecnología.



Bibliografía

[AS03] Jean-Paul Allouche y Jeffrey Shallit. *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press, 2003. DOI: 10.1017/CBO9780511546563 (véase página 36).

[CCS13] C. Calude, G.J. Chaitin y A. Salomaa. *Information and Randomness: An Algorithmic Perspective*. Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2013. ISBN: 9783662030493. URL: https://books.google.com.mx/books?id=PseqCAAAQBAJ (véase página 16).

[Cha66] Gregory J. Chaitin. «On the Length of Programs for Computing Finite Binary Sequences». En: *J. ACM* 13.4 (oct. de 1966), páginas 547-569. ISSN: 0004-5411. DOI: 10.1145/321356.321363. URL: https://doi.org/10.1145/321356.321363 (véase página 15).

[CV05] R. Cilibrasi y P.M.B. Vitanyi. «Clustering by compression». En: *IEEE Transactions on Information Theory* 51.4 (2005), páginas 1523-1545. DOI: 10.1109/TIT.2005.844059 (véase página 19).

[Cop04] B.J. Copeland. *The Essential Turing*. Clarendon Press, 2004. ISBN: 9780191520280. URL: https://books.google.com.mx/books?id=VlC5MkVIwqkC (véase página 15).

[DZ12] Jean-Paul Delahaye y Hector Zenil. «Numerical evaluation of algorithmic complexity for short strings: A glance into the innermost structure of randomness». En: *Applied Mathematics and Computation* 219.1 (2012), páginas 63-77 (véase página 20).

[HH19] Santiago Hernández-Orozco Hector Zenil Liliana Badillo y Francisco Hernández-Quiroz. «Coding-theorem like behaviour and emergence of the universal distribution from resource-bounded algorithmic probability». En: *International Journal of Parallel, Emergent and Distributed Systems* 34.2 (2019), páginas 161-180. DOI: 10.1080/17445760.2018.1448932. eprint: https://doi.org/10.1080/17445760.2018.1448932. URL: https://doi.org/10.1080/17445760.2018.1448932 (véase página 19).

[Kol68] A. N. Kolmogorov. «Three approaches to the quantitative definition of information *». En: *International Journal of Computer Mathematics* 2.1-4 (1968), páginas 157-168. DOI: 10.1080/00207166808803030. eprint: https://doi.org/10.1080/00207166808803030. URL: https://doi.org/10.1080/00207166808803030 (véase página 15).

[Lev74] L. A. Levin. «Laws of information conservation and problems of foundation of probability theory». Russian. En: *Probl. Peredachi Inf.* 10.3 (1974), páginas 30-35. ISSN: 0555-2923 (véase página 16).

[Rad62] T. Rado. «On Non-Computable Functions». En: *Bell System Technical Journal* 41.3 (1962), páginas 877-884. DOI: https://doi.org/10.1002/j.1538-7305.1962.tb00480.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1962.tb00480.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1962.tb00480.x (véase página 17).

[Sha48] Claude Elwood Shannon. «A Mathematical Theory of Communication». En: *The Bell System Technical Journal* 27 (1948), páginas 379-423. URL: http://plan9.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf (visitado 22-04-2003) (véase página 11).

[Sny23] David Snyder. *bzip2*. https://gitlab.com/bzip2/bzip2/. Accessed: (November 21, 2023). 2023 (véase página 14).

[Sol+14] Fernando Soler-Toscano et al. «Calculating Kolmogorov complexity from the output frequency distributions of small Turing machines». En: *PloS one* 9.5 (2014), e96223 (véanse páginas 20, 26).

[Sol64] R.J. Solomonoff. «A formal theory of inductive inference. Part I». En: *Information and Control* 7.1 (1964), páginas 1-22. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(64)90223-2. URL: https://www.sciencedirect.com/science/article/pii/S0019995864902232 (véase página 16).

[Sto15] James V. Stone. *Information Theory: A Tutorial Introduction*. 1st. Sebtel Press, 2015. ISBN: 0956372856; 9780956372857 (véase página 12).

[Tal24] K. Talaga S. Tsampourakis. *PyBDM: Python interface to the Block Decomposition Method (0.1.0)*. [Software]. https://zenodo.org/doi/10.5281/zenodo.10652064. 2024 (véase página 31).

[ZKT17] Hector Zenil, Narsis A. Kiani y Jesper Tegnér. «Low-algorithmic-complexity entropy-deceiving graphs». En: *Phys. Rev. E* 96 (1 jul. de 2017), página 012308. DOI: 10.1103/PhysRevE.96.012308. URL: https://link.aps.org/doi/10.1103/PhysRevE.96.012308 (véase página 41).

[ZKT23] Hector Zenil, Narsis A. Kiani y Jesper Tegnér. *Algorithmic Information Dynamics: A Computational Approach to Causality with Applications to Living Systems*. Cambridge University Press, 2023. DOI: 10.1017/9781108596619 (véanse páginas 21, 23).

[Zen+18] Hector Zenil et al. «A Decomposition Method for Global Evaluation of Shannon Entropy and Local Estimations of Algorithmic Complexity». En: *Entropy* 20.8 (2018). ISSN: 1099-4300. DOI: 10.3390/e20080605. URL: https://www.mdpi.com/1099-4300/20/8/605 (véanse páginas 18, 24).

[ZL78] J. Ziv y A. Lempel. «Compression of individual sequences via variable-rate coding». En: *IEEE Transactions on Information Theory* 24.5 (1978), páginas 530-536. DOI: 10.1109/TIT.1978.1055934 (véase página 14).