Tema 15

Autómatas Celulares Elementales

15.1. Introducción a los autómatas celulares

Los autómatas celulares (AC) son modelos matemáticos discretos que nos permiten simular computacionalmente procesos de gran complejidad en distintos ámbitos de la ciencia: en física, matemáticas, química, biología, ecología, economía, sociología, etc. Estos modelos están compuestos por un conjunto finito y discreto de elementos, simples e idénticos, dispuestos regularmente en el espacio. Cada elemento o "célula" del autómata (de ahí el nombre) puede tomar un número finito y discreto de estados (o valores). Los estados de las células del autómata evolucionan sincrónicamente, esto es, al mismo tiempo, en pasos discretos de tiempo de acuerdo a unas reglas locales que simulan las interacciones del proceso que se quiere simular. El estado de una celda concreta en un momento dado está determinado por los estados pasados de los elementos situados en una determinada vecindad alrededor de la celda, de acuerdo con las reglas de evolución.

Al tratarse de modelos muy intuitivos y bastante fáciles de programar (veremos algunos ejemplos de ello), y al poder disponer de una gran capacidad computacional de cálculo gracias a los modernos procesadores, los autómatas celulares son continuamente utilizados con éxito en la investigación científica, consiguiendo reproducir comportamientos de una complejidad considerable.

La idea que se persigue mediante estos modelos es la siguiente: si conseguimos reproducir un comportamiento complejo a partir de un conjunto de reglas simples, eso puede significar que hemos conseguido capturar la "esencia" de ese fenómeno, es decir, que hemos conseguido descubrir las variables o mecanismos que dan lugar a ese comportamiento. Esto no siempre es así. Sin embargo, cuando lo es, el modelo proporciona una información muy valiosa, no sólo por la comprensión del fenómeno, objetivo primero y fundamental de la investigación teórica, sino por la posibilidad de poder controlar el fenómeno a partir del control de estas reglas para obtener el resultado deseado, objetivo último de la ciencia aplicada.

15.1.1. Características fundamentales de un AC

A continuación describimos en detalle las características básicas que definen a un autómata celular:

1. El espacio es discreto y homogéneo. En un AC, el espacio es dividido en celdas (o "células") consiguiendo una red discreta que puede tener cualquier geometría en cualquier dimensión del espacio $d = 1, 2, 3, \ldots$, incluso puede ser una red fractal como los que hemos visto en el Tema 13. Todas las celdas son iguales, es decir, tienen las mismas propiedades y obedecen a las mismas reglas.

2. Número finito y discreto de estados. Cada celda de la red, que por ejemplo puede representar una partícula o una región del espacio, se encuentra en un estado, y el conjunto de posibles estados es finito y discreto. Ese conjunto de posibles estados es el mismo para todas las células del AC (condición de homogeneidad). Por ejemplo, la celda en la posición *i* está en un determinado estado que denotaremos por a_i, y los valores que puede tomar a_i forman un conjunto contable (discreto y finito). Por ejemplo: $a_i \in \{1,2,3,...,k\}$.

3. El tiempo es discreto. El autómata evoluciona discretamente en el tiempo a través de una serie de pasos temporales, que llamaremos pasos de simulación. En cada paso de la simulación el valor del tiempo se incrementa en una cantidad Δt que en muchos casos es constante. En cada paso de la simulación el estado de cada celda es actualizado dando lugar a una secuencia temporal discreta de estados:

$$... \to a_i[t-\Delta t] \to a_i[t] \to a_i[t+\Delta t] \to ,$$

y esto para todas las celdas.

4. Actualización simultánea. Todas las celdas de la red actualizan su estado al mismo tiempo. Esto quiere decir que en cada paso $t \to t+\Delta t$ se barre toda la red pasando por cada una de las posiciones y actualizando su estado.

5. Reglas de actualización deterministas o estocásticas. Los estados de cada celda son actualizados de acuerdo a unas reglas fijas previamente definidas y que pueden ser *deterministas* (el estado final está unívocamente determinado por el estado inicial) o *estocásticas* (a partir de un mismo estado inicial podemos obtener diferentes estados finales con una cierta probabilidad cada uno).

6. Las reglas son locales en el espacio y en el tiempo. Como característica general, el conjunto $\{R\}$ de reglas utilizadas para actualizar los diferentes estados de cada célula depende del estado de la propia célula y de los estados de las celdas situadas en su vecindad en ese mismo instante y/o en instantes pasados. De este modo, el estado de la celda depende de los estados pasados de ella misma y de su vecindad. Normalmente, la dependencia en el pasado no suele ir más allá de un único paso temporal y la dependencia de la vecindad no extenderse más allá de los *primeros vecinos* (conjunto de celdas más próximas). Por ejemplo, si consideramos una red cuadrada en dos dimensiones (cada celda está referida mediante sus coordenadas de fila *i* y columna *j*, que son números enteros) y definimos la vecindad más simple como las 4 celdas más próximas: arriba, abajo, derecha e izquierda, tendremos que si la dependencia temporal llega sólo hasta el pasado inmediato, la evolución del estado de una célula del autómata tendrá la siguiente forma:

$$a_{i,j}[t+\Delta t] = R\left(a_{i,j}[t], a_{i,j+1}[t], a_{i,j-1}[t], a_{i+1,j}[t], a_{i-1,j}[t]\right).$$

15.1.2. Propiedades de los ACs

Los autómatas celulares son descripciones de fenómenos en los que las ecuaciones son traducidas a reglas, como en un juego. Por consiguiente, describen la naturaleza de un modo muy intuitivo, fácilmente describible y comprensible. Además, son de fácil diseño y programación. Esa simplicidad hace que se puedan compilar y ejecutar los códigos muy rápidamente, especialmente porque los algoritmos de autómata celular son altamente paralelos, por lo que son muy propicios para la programación en paralelo (aquella en la que se utilizan varios procesadores de forma simultánea para ejecutar un mismo programa). Eso permite que podamos simular grandes redes, con muchos elementos, para de ese modo poder estudiar su comportamiento *macroscópico* (comportamiento promedio sobre un gran

número de elementos) para tiempos muy largos, mucho más largos que los que se puede estudiar mediante otros importantes métodos de simulación como *la dinámica molecular*. Sin embargo, se corre el peligro de que, en la búsqueda de esa simplicidad, la física del mecanismo que se simula sea sacrificada en exceso, pudiendo desobedecer posibles leyes físicas.

15.1.3. Aplicaciones de los ACs

Hay un sinfín de procesos naturales que pueden ser modelados mediante autómatas celulares.

Las plantas, por ejemplo, regulan la fijación y expulsión de gases a través de un mecanismo de AC, cada estoma de la hoja actuando como una celda. Las *redes neuronales* también pueden ser interpretadas como autómatas celulares. Los complejos y cambiantes patrones ondulatorios observados sobre la piel de los cefalópodos o los patrones de activación en los cerebros humanos pueden ser simulados mediante ACs.

Como es obvio, los autómatas celulares tienen una aplicación directa en sistemas que contienen un gran número de elementos similares con interacciones locales. En esta línea, la *física de superficies* se presenta como una de las principales aplicaciones de los autómatas celulares; las celdas suelen representar los átomos de la red cristalina de un sólido y los estados de las celdas pueden identificar el tipo de átomo, la especie química que se encuentra sobre la superficie o el valor de cualquier otro observable físico-químico. De este modo se pueden simular computacionalmente reacciones químicas superficiales (de gran importancia en procesos de *catálisis*, por ejemplo) o procesos en los que el sólido crece o se desintegra (procesos de *crecimiento de superficies*) como la *deposición*, la *agregación*, la *solidificación*, la *erosión* o la *corrosión* de metales. En las Figs. 15.1 y 15.2 se muestran un par de ejemplos.

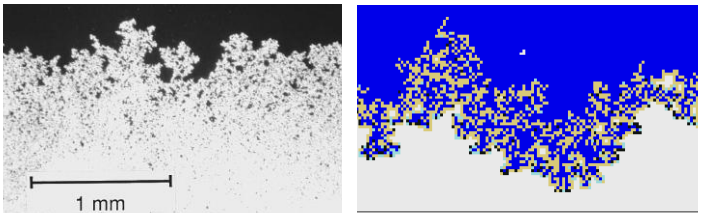


Figura 15.1: Izquierda: foto de la corrosión en una lámina de aluminio. Derecha: frente de corrosión obtenido a partir de un autómata celular

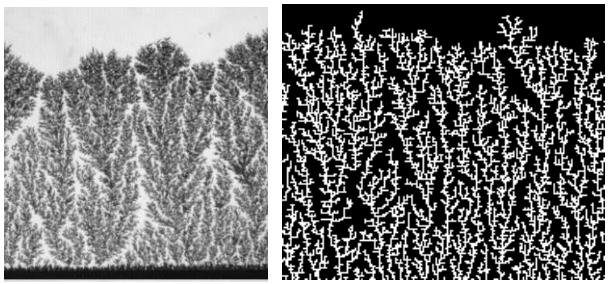


Figura 15.2: Izquierda: crecimiento por electrodeposición de plata. Derecha: simulación computacional mediante un autómata celular.

Otro ejemplo de aplicación directa de los ACs es la dinámica de poblaciones en biología y ecología, en la que cada elemento del autómata representa a un individuo o grupo de individuos de una o varias poblaciones con interacciones ecológicas entre ellos y con el medio. Tal es el caso de los conocidos modelos depredador-presa (entre ellos los más conocidos están los *modelos de Lotka-Volterra*).

Una aplicación muy importante de la simulación mediante autómatas celulares son los *procesos de reacción-difusión*, en los que tenemos un conjunto de elementos que difunden y reaccionan entre sí.[1] Estos procesos pueden dar lugar a la formación de estructuras y patrones complejos en sistemas biológicos. Por ejemplo, los patrones de pigmentación observados sobre la piel de numerosas especies animales son generados a partir de funciones biológicas que están gobernadas por sistemas químicos de reacción-difusión, esto es, sistemas en los que tenemos un conjunto de especies químicas que difunden en una solución y reaccionan químicamente entre sí. Cada celda representa una célula y sus posibles estados etiquetan típicamente el tipo de célula o el estado de su evolución (o de su pigmentación, por ejemplo). Las formaciones vegetales en forma de bandas de distintos grosores, parches circulares o laberintos, observadas en medios áridos y semiáridos, y cuya investigación es de vital importancia para comprender como reacciona la naturaleza ante el proceso de desertificación, también pueden ser modelizadas mediante sistemas de reacción-difusión. Estas formaciones aparecen muy a menudo gobernadas por reglas locales que pueden fácilmente ser implementadas en un autómata celular. En este caso, las celdas de la red representan áreas superficiales o parches, mientras que los valores que toma cada célula indicarían la cantidad de biomasa que contiene. En la Fig. 15.3 mostramos algunos ejemplos de estos patrones y el resultado de su simulación mediante autómatas celulares.

[1] Imagínese el caso de un gas compuesto por moléculas que difunden aleatoriamente y reaccionan químicamente entre sí cuando entran en contacto.

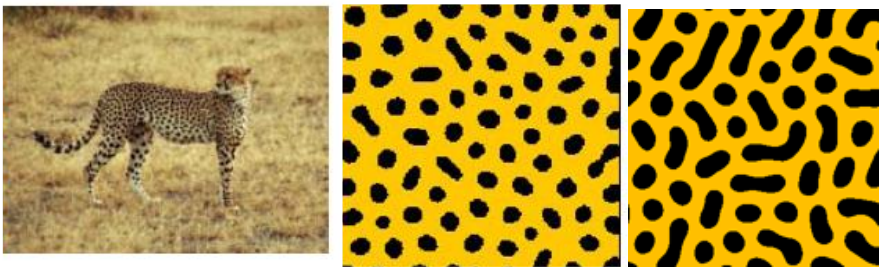


Figura 15.3(a): Izquierda: diferentes patrones observados en la piel de algunas especies animales, y algunos ejemplos de formaciones vegetales observadas en medios áridos-semiáridos. Derecha: patrones obtenidos de la simulación de un sistema de reacción-difusión mediante autómatas celulares. Es interesante darse cuenta de la universalidad de los patrones, lo que indica que detrás de todos ellos subyace el mismo tipo de fenómeno.

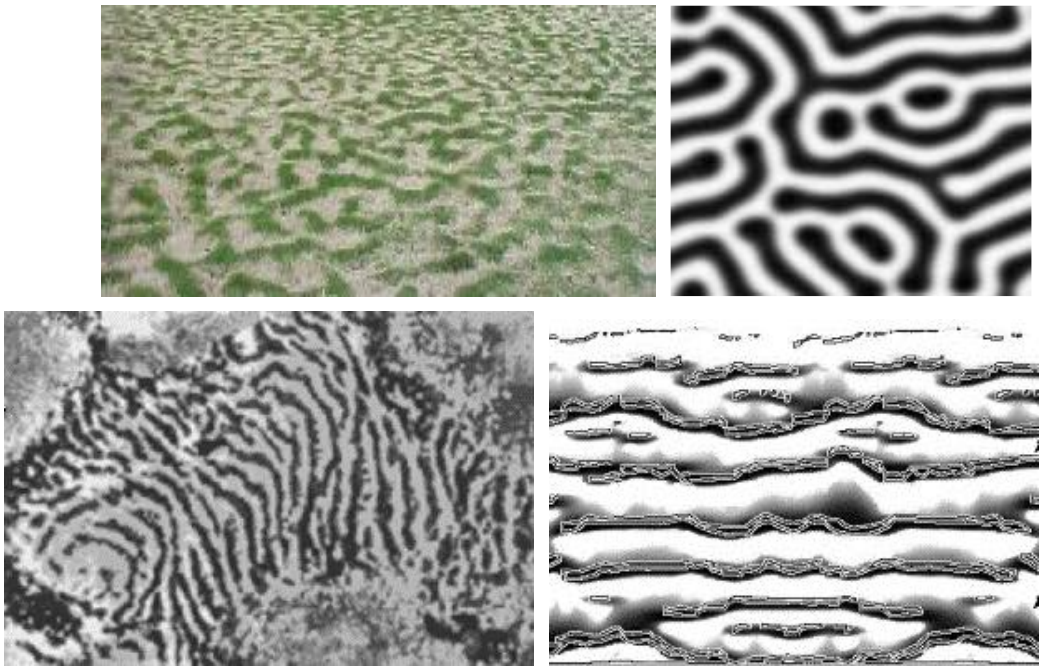


Figura 15.3(b): Izquierda: ejemplos de formaciones vegetales observadas en medios áridos-semiáridos. Arriba: estructuras laberínticas de unos pocos centímetros de grosor en Negev (Israel). Abajo: formación de bandas de vegetación (“tiger bush”) de varias decenas de metros de espesor sobre las laderas de las colinas en Niamei (Nigeria). Derecha: patrones obtenidos de la simulación de un sistema de reacción-difusión mediante autómatas celulares. Es interesante darse cuenta de la universalidad de los patrones, lo que indica que detrás de todos ellos subyace el mismo tipo de fenómeno.

Más adelante estudiaremos uno de los ejemplos más importantes de sistemas de reacción-difusión: los *medios excitables*. Como veremos, los ACs nos permiten simular fácilmente la propagación de ondas en medios excitables. Tal es el caso, por ejemplo, de la propagación de señales nerviosas en el corazón o en el tejido neuronal, la propagación de ondas en reacciones químicas como la reacción de Belousov-Zhabotinsky, o la propagación de una epidemia sobre una población de individuos.

Además de los patrones de pigmentación, los autómatas celulares se han utilizado en otros procesos de *morfogénesis biológica*. Por ejemplo, para simular el crecimiento

ramificado de vasos en los capilares sanguíneos y en las hojas. Véase, por ejemplo, la simulación del crecimiento de las venas en una hoja de Ginkgo mediante un desdoblamiento dicotómico, en la Fig. 15.4, o la simulación del crecimiento ramificado lateral de vasos de la Fig. 15.5, observado en la tráquea de los insectos o en la venación de una hoja de hiedra, por ejemplo.

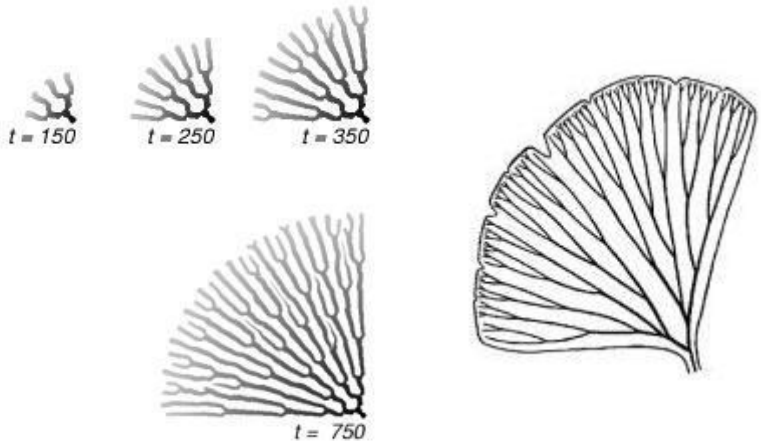


Figura 15.4: Izquierda: simulación mediante un AC del crecimiento de vasos (tamaño del dominio 300×300 celdas). Derecha: hoja real del Ginkgo.

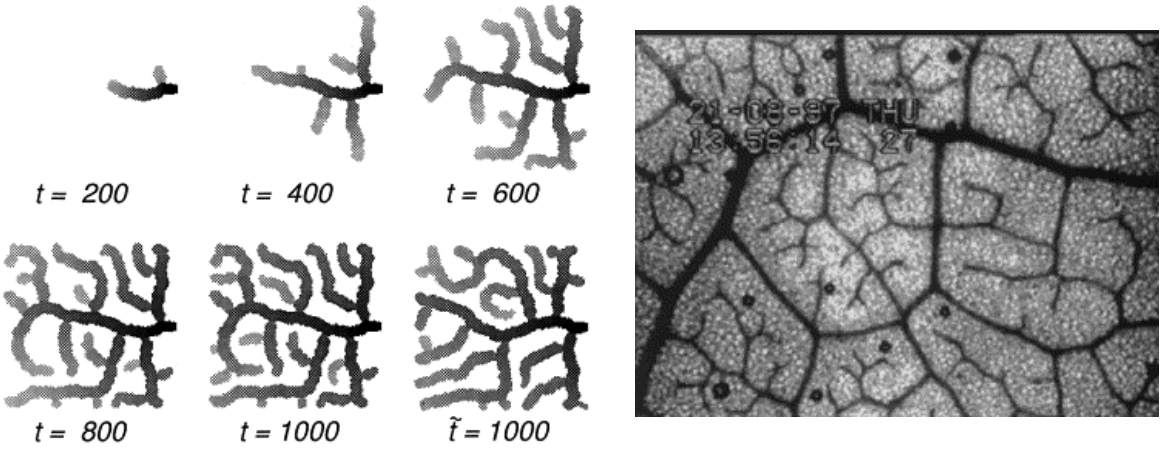


Figura 15.5: Izquierda: simulación mediante ACs del crecimiento ramificado lateral de venas (tamaño del dominio 150×150 celdas). Derecha: fotografía del patrón de venación de una hoja de hiedra.

Los autómatas celulares también se utilizan en áreas de investigación tan importantes como la *física de fluidos* (denominados *lattice gas celular automata*), para estudiar problemas tan complejos como la *turbulencia*. En *aerodinámica*, por ejemplo, se utilizan para simular túneles de viento.

El rango de aplicación de los autómatas celulares como modelos de simulación no tiene límites. En *mecánica estadística* se utilizan para estudiar *transiciones de fase*, *comportamiento auto-organizado* o problemas de *percolación*. En *astrofísica* para simular, por ejemplo, la evolución de galaxias espirales. En *teoría de la computación* para el *procesamiento de datos*, la *imagen digital* o la *visión artificial*.

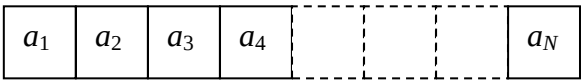
15.2. Autómatas celulares elementales

Los autómatas celulares son esencialmente *sistemas dinámicos* discretos.[2] Al tratarse de sistemas dinámicos muy simples, los ACs han sido utilizados para investigar importantes fenómenos, como por ejemplo, cómo se produce la *auto-organización* en *mecánica estadística*. La mecánica o física estadística es una importante área de la física cuyo objetivo principal es determinar, mediante leyes estadísticas, el comportamiento promedio –o termodinámico– de sistemas macroscópicos (esto es, sistemas grandes o con muchos elementos) a partir del comportamiento microscópico de los elementos del sistema, es decir, el que se produce en la escala más pequeña. La auto-organización es, como su nombre indica, el proceso de evolución espontánea de determinados sistemas hacia un comportamiento ordenado no esperado.

En este capítulo vamos a presentar los autómatas celulares más sencillos, denominados *autómatas celulares elementales* (ACE), y los vamos a utilizar para describir importantes propiedades y características de los sistemas dinámicos y de su descripción mediante la mecánica estadística.

Las características de estos autómatas son:

- Dimensión 1 ($d=1$). Estos autómatas consisten en una fila de *N* celdas en la que cada celda está determinada por su posición *i* en la red: $i=1,...,N$. El estado de cada celda será a_i:



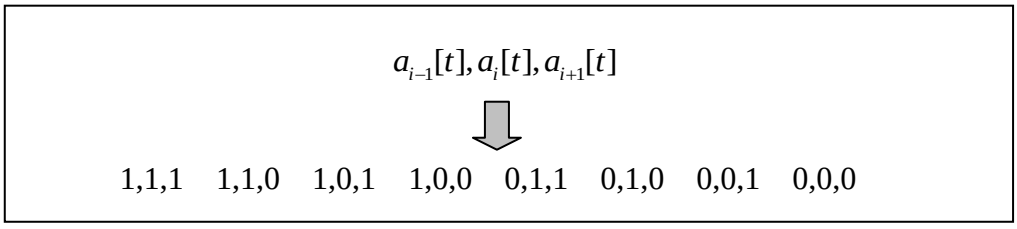
- Cada celda puede estar en dos estados: 0 (celda vacía) y 1 (celda ocupada):

$$a_i[t] \in \{0,1\}.$$

- La vecindad de una celda abarca a la propia celda y a sus dos celdas inmediatamente vecinas, la de su izquierda y la de su derecha (en el argot de la simulación es lo que se llama *primeros vecinos*, en inglés *nearest neighbours*, NN).
- El tiempo avanza en pasos discretos de una unidad, y el estado de la celda en el tiempo $t+1$ depende del estado de su vecindad en el tiempo *t* a través de una regla de evolución *R*:

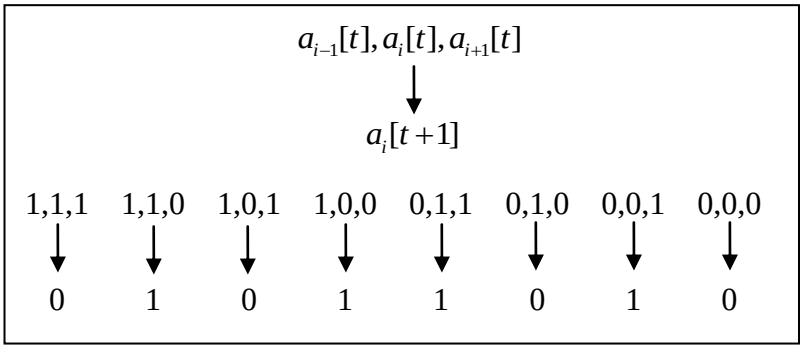
$$a_i[t+1] = R\left(a_{i-1}[t], a_i[t], a_{i+1}[t]\right)$$

- Como hemos visto, la regla *R* de actualización del estado de cada celda es una función de los estados de la vecindad. Como cada celda puede estar en dos estados, las posibles vecindades diferentes que puede tener una celda serán 8. Estas configuraciones son:



[2] Los sistemas dinámicos discretos se estudiaron en el Tema 14.

Por consiguiente, la regla R de evolución de nuestro autómata celular elemental vendrá determinada por 8 dígitos que pueden tomar el valor 0 o 1. Consideremos por ejemplo la regla $R = 01011010$. Esto quiere decir que

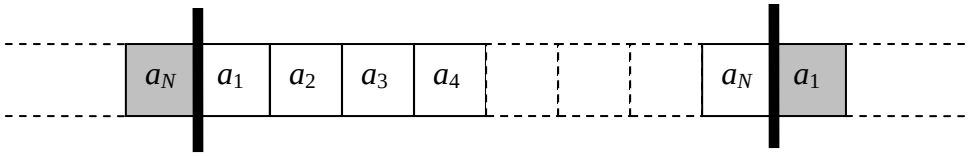


Como cualquier secuencia de 8 dígitos binarios determina una regla, tenemos 2^8 reglas posibles, es decir, 2^8 variaciones con repetición posibles de 2 elementos tomados de 8 en 8. En concreto, la regla que hemos utilizado como ejemplo (denominada regla 90 porque el número 90 expresado en base dos tiene la forma 01011010: $90 = 0\times 2^7 + 1\times 2^6 + 0\times 2^5 + 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0$) se obtiene de la operación booleana (lógica) denominada *disyunción exclusiva*:

$$R_{90}: \quad a_i[t+1] = a_{i-1}[t] \oplus a_i[t]$$

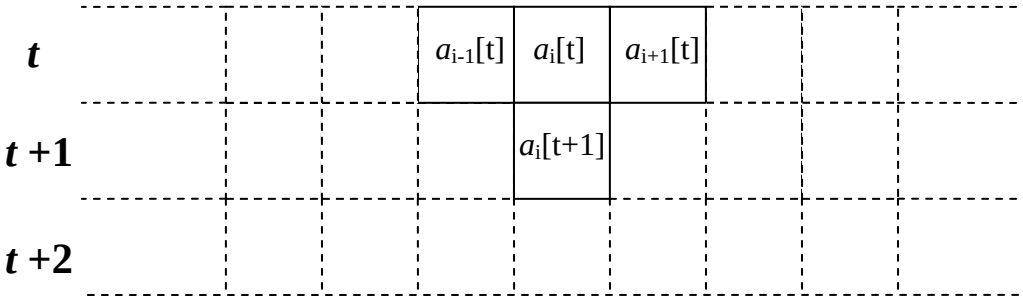
Esta regla también se denomina *suma módulo 2* porque su resultado es el mismo que el que se obtiene al sumar los valores de los estados de las dos celdas vecinas de los lados, dividirlos por dos y devolver el resto de la división. Muchas de las 2^8 posibles reglas pueden ser expresadas como funciones booleanas de los estados de las celdas dentro de la vecindad.

De esta forma, partiendo de una configuración inicial, esto es, un estado inicial en el que todas las celdas de la fila se encuentran en un determinado estado 0 o 1, podemos hacer evolucionar nuestro autómata paso a paso actualizando los estados de cada una de las celdas de acuerdo con una determinada regla. Obsérvese que para actualizar las celdas de los extremos: a_1 y a_N, no tenemos celda vecina izquierda ni celda vecina derecha respectivamente. Para resolver esto lo que se suele hacer en simulación computacional es elegir unas condiciones de contorno, lo que implica definir los estados de los extremos de la fila. Las condiciones de contorno más utilizadas son las *condiciones de contorno periódicas* (en inglés *periodic boundary conditions*), que consisten en suponer que la red se repite a la izquierda y a la derecha como se ilustra en la figura



Esto es análogo a suponer que la fila de celdas se dobla formando una circunferencia.

Supongamos ahora que “desplegamos” la evolución temporal del autómata, es decir, que a medida que se suceden los pasos de simulación vamos colocando la fila correspondiente a cada paso de la evolución debajo de la anterior, como se muestra en la siguiente figura:



Desde un punto de vista algorítmico, esto significa que estamos considerando una red cuadrada en la que cada fila está denotada por *t* y cada columna por *i*. Entonces tenemos que $a_{t,i} = a_i[t]$. La simulación comienza definiendo el estado inicial (fila $t = 0$), es decir, dando valores iniciales a las celdas de la fila superior $a_{0,i}$. En cada paso *t* de la simulación barremos con un bucle la fila *t*, utilizando un for por ejemplo: $\text{for (i=1; i} \le N\text{; i++)}$, y actualizamos los estados de las celdas de esa fila aplicando la regla de evolución sobre las vecindades de cada celda en la fila $t-1$.

En la Fig. 15.6 se muestra el resultado de este proceso para diferentes reglas de evolución que parten de la misma configuración inicial simple. El estado inicial consiste en una semilla, es decir, una configuración en la que todas las celdas de la fila superior se encuentran en estado 0 excepto una (situada en el medio) que se encuentra en estado 1. En la figura se ha mostrado la evolución de la simulación hasta el momento en el que la configuración se repite (se dice que el sistema ha alcanzado un *estado estacionario*), como por ejemplo en las reglas 0, 4, 32, 36, 72, 76, 104 y 108, y cuando no es el caso se han mostrado los 20 primeros pasos de simulación. El proceso es análogo al crecimiento de un cristal a partir de una semilla microscópica.

Como se puede comprobar, a partir de una condición inicial extremadamente simple y sin ninguna estructura, y de reglas de evolución locales muy simples, obtenemos una gran variedad de comportamientos, algunos de ellos muy complejos. Recordamos que en todos los casos la localidad espacial se extiende sólo los primeros vecinos, mientras que la localidad temporal se reduce al pasado más cercano (el paso anterior).

Podemos distinguir dos tipos de reglas o comportamientos:

-Reglas simples que dan lugar a estados estacionarios (estados que no evolucionan en el tiempo), como las reglas 0, 4, 32, 36, o a estados homogéneos con patrones uniformes triviales (como las reglas 50, 54, 94, 222…).

-Reglas complejas que dan lugar a patrones fractales autosimilares. Tal es el caso de la regla 90, que genera el Triángulo se Sierpinsky ($d_f = 1.59$) que ya estudiamos en el Tema de Fractales, y que generamos con un algoritmo que escondía esta regla sin saber entonces que estábamos implementando un autómata celular. O la regla 150, por ejemplo, que da lugar a otro fractal con $d_f = 1.69$. En estas estructuras fractales, el número *T* de triángulos observados con un cierto tamaño *n* obedece la siguiente *ley de potencias*:

$$T(n) \sim n^{-d_f} \quad \text{(auto-similaridad)}$$

Estas leyes de potencias aparecen siempre en los fractales y reflejan la invariancia bajo cambios de escala de estas estructuras (característica intrínseca a los fractales).

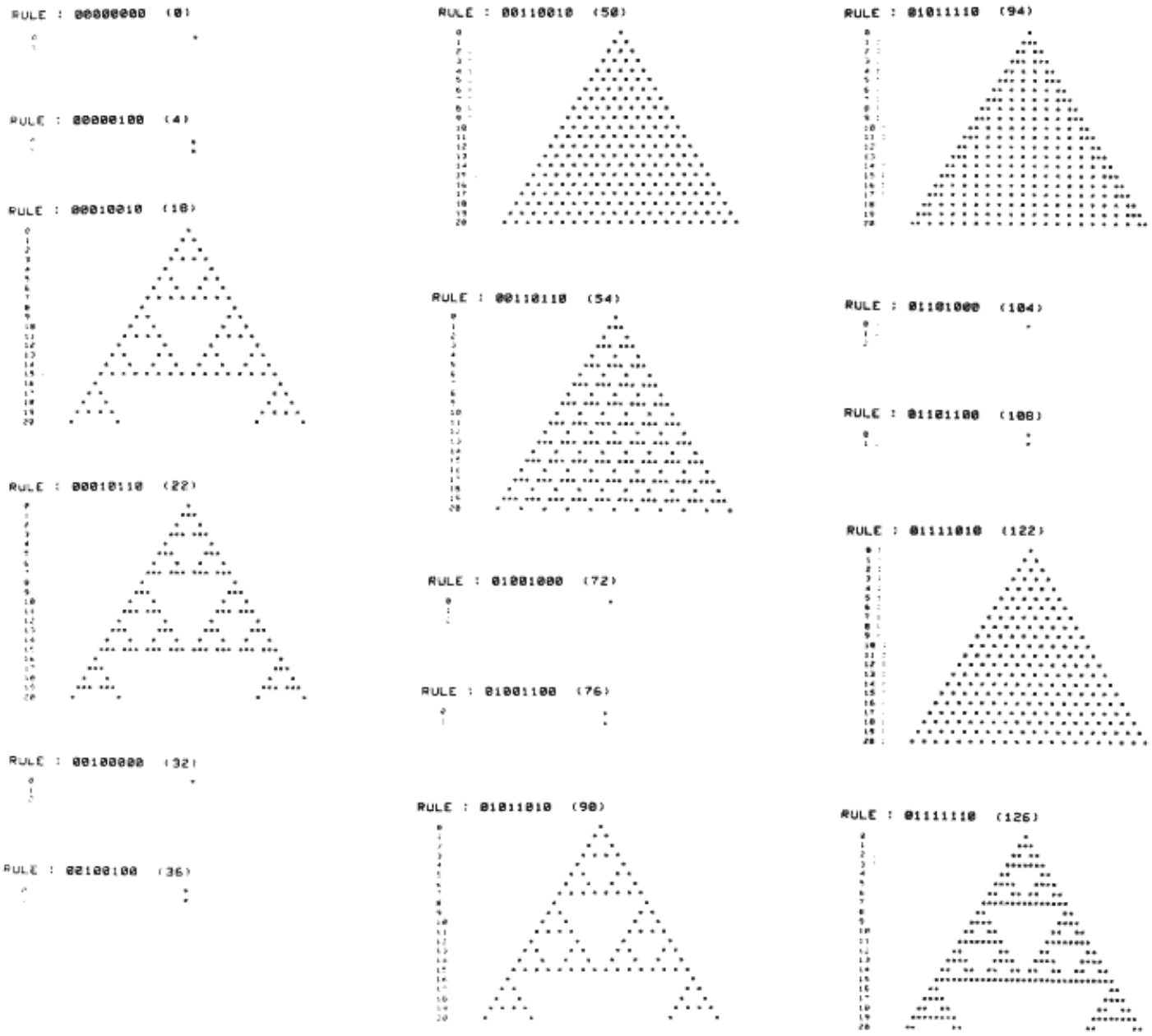


Figura 15.6: Evolución de autómatas celulares elementales unidimensionales con diferentes reglas que empiezan en un estado inicial que contiene una única posición con valor 1. Las celdas con estado 1 han sido representadas mediante puntos mientras que aquellas con estado 0 son mostradas con espacios en blanco. La configuración de cada autómata celular en pasos sucesivos es mostrada en líneas sucesivas.

A continuación vamos a ver cómo podemos simular la evolución de un ACE a partir de una regla dada. En este tipo de autómatas, los estados de las celdas, las vecindades y las reglas de evolución se representan con ceros y unos, por lo que pueden ser tratados directamente como números binarios. Vamos a ver que el cálculo de una regla a partir de su número, y la aplicación de esta regla a una vecindad dada se implementa de manera natural en C usando máscaras binarias, operadores lógicos y el "*bit shift*" (desplazamiento de bits).

Las posibles vecindades (de primeros vecinos) que tiene una celda pueden numerarse del 0 al 7 a partir de su codificación binaria. Así, cuando $a_{i-1}=1$, $a_i=1$, $a_{i+1}=0$, se trata de la vecindad 6 ya que este número tiene en binario la representación 110.

Nota: Compruébese que la representación binaria del 6 es 110 mediante el código

printf ("%d\n", 0B110);

Aquí se ha introducido la notación numérica binaria del C, que consiste en preceder la expresión binaria con un "0B" a la izquierda.

A su vez, la expresión binaria de la regla del ACE proporciona de forma ordenada el resultado de aplicar la regla a estos 8 diferentes vecindarios: el primer dígito de la regla corresponde al resultado de la regla sobre la vecindad 7 (111 en binario), el segundo dígito es el resultado sobre la vecindad 6 (110 en binario) y así sucesivamente hasta el último dígito, que es el resultado de aplicar la regla sobre la vecindad 0 (000 en binario).

Partiendo del número de la regla y del número de vecindario vamos a obtener un algoritmo eficiente que dé el resultado de la aplicación de la primera sobre el segundo. Para ello, lo primero es componer el número de vecindario a partir de los tres estados de la vecindad (a_{i-1}, a_i, a_{i+1}). Hemos visto que la expresión algebraica es

$$n_V = 2^2 \times a_{i-1} + 2^1 \times a_i + 2^0 \times a_{i+1}$$

Dado que cada estado puede ser 0 o 1, que son dígitos binarios, en C lo más fácil es utilizar los operadores " << " y " | " para ensamblar el número n_V. El código es:

$n_V = (a[i+1] \mid a[i] << 1 \mid a[i-1] << 2)$;

Lo que hace este código es desplazar los bits de los estados (los valores de los *a*) a su posición final en la expresión binaria del número de vecindario usando el operador " << " (a cuya derecha figura el número de puestos que los bits se desplazan hacia la izquierda). Luego se unen estos tres valores en un único número mediante el operador binario " | ", que opera bit a bit de un número entero.

A continuación, sabiendo el número de vecindario, simplemente se busca el bit adecuado dentro de la expresión binaria de la regla. Para ello volvemos a utilizar el operador de desplazamiento: desplazamos la expresión binaria de la regla tantos puestos hacia la derecha como bits queremos examinar (esto es, como número de vecindario queramos), y luego leemos el valor de ese bit "enmascarándolo" con el 1, usando para ello el operador binario " & ". El código sería:

$\mathrm{r} = (\mathrm{REGLA} >> n_V) \ \& \ 1;$

Nota: Otra forma de hacer lo mismo, pero con cierta complicación, sería desplazando el bit 1 hasta la posición del número de vecindario (usando " << "), hacer la máscara con " & " allí para leer el bit de la regla, y luego traerlo a la derecha de la expresión binaria deshaciendo el desplazamiento (ahora con " >> "). El código sería entonces:

$\mathrm{r} = \left((1 << n_V) \ \& \ \mathrm{REGLA}\right) \ >> \ n_V;$

Este código es menos eficiente que el anterior porque usa dos "*bit shifts*" en vez de uno.

En el listado 15.1(a) se muestra el código del programa que simula la evolución de un ACE de acuerdo con una regla definida, comenzando en una configuración inicial ($t = 0$) que consiste en una semilla (una única celda en estado 1) situada en el punto medio de la fila inicial. Obsérvese que al final de cada paso de evolución se actualizan las celdas laterales $j = 0$ y $j = N + 1$, ya que son las que simulan las condiciones de contorno periódicas.

El programa genera una imagen como las mostradas en la Fig. 15.6. Para ello se mandan los datos del resultado de la evolución del ACE a una función como la definida en la Sec. 11.10 del Tema 11, guardaPGMi ( ),[3] que construye una imagen en escala de grises y la guarda en un archivo con formato PGM. En nuestro caso sólo tendremos dos colores ya que sólo tenemos dos estados: 0 (blanco) y 1 (negro). Esta función ha sido recopilada en forma de biblioteca, libguardaimagen.c, véase listado 15.1(b), para que la podamos utilizar siempre que queramos independientemente del programa que genere la matriz de píxeles, y se encuentra declarada en el archivo H <libguardaimagen.h>, mostrado en el listado 15.1(c).

Listado 15.1(a): Programa que simula la evolución de un ACE según la regla definida, con una configuración inicial que consiste en una semilla: una única celda en estado 1 (en el medio) y el resto en estado 0.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "libguardaimagen.h"

#define REGLA 54
#define N 1000 /* número de celdas del ACE */
#define T 500 /* número de pasos de evolución */

int main(int argc, char** argv)
{
        int i, j, nv;
        int ACE [ T+1 ][ N+2 ];

        /* definimos la condición inicial de nuestro ACE */
        for ( i = 0; i < T+1; i ++)
        {
                for  ( j = 0; j < N+2; j ++)
                {
                        ACE [ i ][ j ] = 0;
                }
        }
        ACE [ 0 ][ N/2 ] = 1;    /*ponemos la semilla */

        /* simulamos los T primeros pasos de la evolución del ACE */
        for ( i = 1; i < T+1; i ++)
        {
                for  ( j = 1; j < N+1; j ++)
                {
                        nv = (ACE [ i-1 ][ j+1 ] | ACE[ i-1 ][ j ] <<1 | ACE [ i-1 ][ j-1 ] <<2);
```

[3] La única diferencia entre la función guardaPGMi ( ) y la función guardaPGMd ( ) es que la primera recibe un valor entero por cada píxel mientras que en la segunda son reales.

```
                    /* también podíamos haber utilizado la expresión:
                    nv = (2^2)*ACE [ i-1 ][ j-1 ] + (2^1)*ACE [ i-1 ][ j ] + (2^0)*ACE [ i-1 ][ j+1 ]; */

                    ACE [ i ] [ j ]= (REGLA >> nv) & 1;
                }
                /* actualizamos las condiciones periódicas de contorno */
                ACE [ i ][ 0 ] = ACE [ i ][ N ];
                ACE [ i ][ N+1 ] = ACE[ i ][ 1 ];
        }

        guardaPGMi("ACE_R54_semilla.pgm", N+2, T+1, (int*)ACE, 1, 0);

        return 0;
}
```

Listado 15.1(b): libguardaimagen.c. Contiene las definiciones de las funciones que guardan una imagen en un archivo con formato PGM.

```
#include <stdio.h>

void guardaPGMi(char* nombre, int anchura, int altura, int *pixels,
                int pixel_min, int pixel_max)
{
        int i, j, ij, p;
        FILE* imagen;

        imagen = fopen (nombre, "wb");
        fprintf (imagen, "P2");
        fprintf (imagen, "#guardaPGMi %s\n", nombre);
        fprintf (imagen, "%d %d\n", anchura, altura);
        fprintf (imagen, "255\n");

        ij=0;
        for (i=0; i<altura; i++)
        {
                for (j=0; j<anchura; j++)
                {
                        p= (255*(pixels[ ij ]-pixel_min)) / (pixel_max-pixel_min);
                        if ( p<0  ) p=0;
                        if ( p>255) p=255;

                        fprintf (imagen, " %d", p);

                        ij++;
                }
                fprintf (imagen, "\n");
        }

        fclose (imagen);

        return;
}
```

```
void guardaPGMd (char* nombre, int anchura, int altura, double *pixels,
                 double pixel_min, double pixel_max)
{
        int i, j, ij, p;
        FILE* imagen;

        imagen= fopen(nombre, "wb");
        fprintf (imagen, "P2");
        fprintf (imagen, "#guardaPGMi %s\n", nombre);
        fprintf (imagen, "%d %d\n", anchura, altura);
        fprintf (imagen, "255\n");

        ij=0;
        for (i=0; i<altura; i++)
        {
                for (j=0; j<anchura; j++)
                {
                        p= (int) (255*(pixels [ ij ]-pixel_min)) / (pixel_max-pixel_min);
                        if ( p<0  ) p=0;
                        if ( p>255) p=255;

                        fprintf (imagen, " %d", p);

                        ij++;
                }
                fprintf (imagen, "\n");
        }

        fclose (imagen);

        return;
}
```

Listado 15.1(c): <libguardaimagen.h>. Contiene las declaraciones de las funciones que guardan una imagen en un archivo con formato PGM.

```
#ifndef _LIBGUARDAIMAGEN_H_
#define _LIBGUARDAIMAGEN_H_

#include <stdio.h>

/* Guarda en el archivo de nombre dado una imagen PGM
* de dimensiones anchura X altura que contiene en sus
* píxeles valores enteros entre pixel_min y pixel_max
*/
void guardaPGMi (char* nombre, int anchura, int altura,
                 int *pixels, int pixel_min, int pixel_max);

/* Guarda en el archivo de nombre dado una imagen PGM
* de dimensiones anchura X altura que contiene en sus
```

```
* píxeles valores reales entre pixel_min y pixel_max
*/
void guardaPGMd (char* nombre, int anchura, int altura,
                 double *pixels, double pixel_min, double pixel_max);

#endif
```

Ejercicio 15.1. Mostrar la imagen generada utilizando el listado 15.1 para las reglas de evolución 4, 54, 90, 150 y 182, considerando los 500 primeros pasos de un ACE compuesto por 1000 celdas.

Analicemos ahora lo que ocurre cuando partimos de una configuración inicial desordenada. En la Fig. 15.7 se muestra el resultado obtenido cuando consideramos un estado inicial perfectamente desordenado, es decir, cuando el valor de cada una de las celdas de la fila inicial es 1 o 0 con probabilidad 1/2. De nuevo se ha mostrado la evolución hasta que una configuración particular aparece por segunda vez, es decir, se repite, o hasta los 30 primeros pasos de la simulación. Al igual que en el caso anterior en el que partíamos de una semilla inicial, aparecen varias clases de comportamiento. De nuevo podemos distinguir dos comportamientos principales:

-Reglas simples que dan lugar a un comportamiento trivial. Con estas reglas la evolución alcanza bien una configuración[4] estacionaria final que no varía con el tiempo (en teoría de sistemas dinámicos esto recibe el nombre de *punto fijo, punto límite* o *punto estable*), como en las reglas 0, 4, 32, 36, 72, 76, 104, 128 y 132, o bien un comportamiento periódico, o *ciclo límit*e, en el que los estados del ACE se repiten periódicamente al cabo de un cierto número de pasos; en los casos mostrados en la figura (reglas 50, 94 y 108) tenemos que los ciclos están compuestos de dos estados, es decir, tienen un periodo de 2. A medida que *N* crece, en el caso de estas reglas simples los autómatas celulares elementales generalmente evolucionan hacia ciclos cuyo periodo permanece constante o crece lentamente. Incluso ACs elementales infinitos pueden dar lugar a un comportamiento periódico para una amplia clase de estados iniciales.

-Reglas complejas que dan lugar a un comportamiento no trivial, *caótico*. Por un lado nos encontramos que los ciclos son mucho más largos y que su periodo crece ilimitadamente a medida que *N* aumenta (en teoría de sistemas dinámicos a esto se le denomina *atractor extraño* y está íntimamente ligado al c*omportamiento caótico* o *caos*). En el caso de las reglas complejas, los ciclos puros sólo ocurren para condiciones iniciales excepcionales. Por otro lado, observamos que la evolución de las reglas complejas da lugar a una cierta *auto-organización* que no existía en el estado inicial perfectamente desordenado (aleatorio) y por tanto no correlacionado. Esta auto-organización produce *correlaciones* entre celdas distantes: el estado de una celda está relacionado con el estado de otras celdas separadas a una cierta distancia denominada *longitud de correlación*. Este orden se manifiesta en forma de triángulos observables en todas las escalas, esto es, triángulos de todos los tamaños, aunque en este caso no produzcan un patrón fractal puesto que obedecen la siguiente relación

$$T(n) \sim \lambda^{-n} \text{ (no auto-similaridad),}$$

lo cual significa que, aunque aparecen triángulos en todas las escalas, no hay invariancia de escala.

[4] Cuando nos refiramos a una configuración particular del ACE, es decir, a una configuración en la que cada celda de la red se encuentra en un determinado estado, utilizaremos también el término “estado del ACE”.

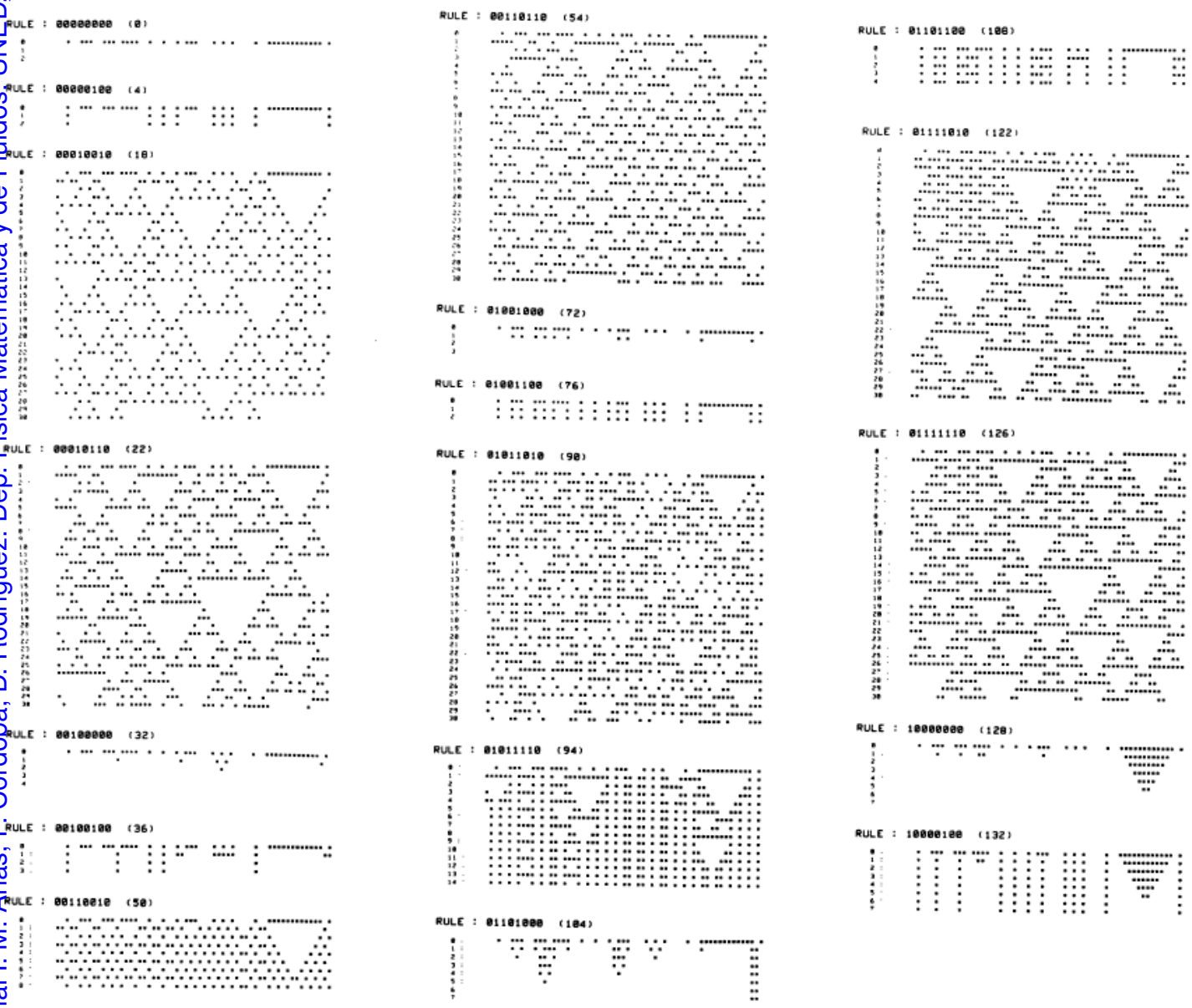


Figura 15.7: Evolución de autómatas celulares elementales unidimensionales con diferentes reglas que empiezan en un estado inicial aleatoriamente desordenado. Las celdas con estado 1 han sido representadas mediante puntos mientras que aquellas con estado 0 son mostradas con espacios en blanco. La configuración de cada autómata celular en pasos sucesivos es mostrada en líneas sucesivas.

Ejercicio 15.2. El objetivo de este ejercicio es simular la evolución de un ACE a partir de una configuración inicial aleatoriamente desordenada. La única diferencia con el algoritmo anterior reside en la condición inicial: en este caso se tratará de una fila de unos y ceros repartidos aleatoriamente. Para conseguir esta aleatoriedad necesitamos un generador de número aleatorios. En el Tema 12. Métodos de Monte Carlo, se mostraron distintos algoritmos que generaban números (pseudo-)aleatorios con una cierta distribución de probabilidad, en concreto se vieron dos casos: números aleatorios uniformemente distribuidos en el intervalo [0,1) y números aleatorios con una distribución gaussiana (aunque en realidad estos se obtenían a partir de los primeros). El método para generarlos se denominaba método congruente lineal.

Afortunadamente no va ser necesario programar este método pues el lenguaje C cuenta con una función que ya lo hace por nosotros: rand ( ), declarada con la siguiente sintaxis:

int rand (void)

y definida en la librería <stdlib.h>. Esta función no tiene argumentos y devuelve un número entero uniformemente distribuido en el intervalo $[0, \mathrm{RAND\_MAX}]$, donde RAND_MAX es el máximo valor generado, y que suele ser $\mathrm{RAND\_MAX} = 2^{31} - 1 = 2147483647$. Esto quiere decir que la llamada a rand( ) devolverá cualquier número entero dentro de ese intervalo con la misma probabilidad.

Previamente se tiene que haber definido (dentro de la función main, por ejemplo) la semilla inicial del generador de números aleatorios mediante la función srand( ), declarada con la siguiente sintaxis:

void srand (unsigned int semilla)

Si esta semilla se ha definido a partir de una constante entera, cada vez que se ejecute el programa la secuencia de números pseudo-aleatorios generada por rand( ) será siempre la misma. Esto quiere decir que las secuencias son reproductibles, es decir, podemos repetir la misma secuencia de números aleatorios, por grande que sea, cuantas veces queramos para estudios comparativos. Aunque este último aspecto resulta de gran utilidad, deja en clara evidencia que no se trata de números estrictamente aleatorios. Por el contrario, se puede definir esa semilla a partir de una variable que cambie con el tiempo. Si escribimos por ejemplo:

srand (time (NULL));

la semilla estará referida al reloj de la CPU, y cada vez que se ejecute el programa la semilla será distinta y también lo será la secuencia de números pseudo-aleatorios generados.

A continuación mostramos dos ejemplos. En el primero de ellos el programa imprime en pantalla los diez primeros números pseudo-aleatorios de la secuencia generada a partir de la semilla 20. Podemos comprobar que cada vez que se ejecuta el programa, la secuencia es la misma. En el segundo, el programa realiza la misma tarea con la diferencia de que la semilla cambia dinámicamente, por lo que cada vez que se ejecuta el programa la secuencia varía.

Ejemplo 1

```
#include <stdio.h>
#include <stdlib.h>


int main (int argc, char** argv)
{
    int i, semilla=20;
    srand (semilla);
    for (i=0; i<10; i++)
        printf("%d,\t%d\n", i, rand( ));

    return 0;
}
```

Ejemplo 2

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (int argc, char** argv)
{
    int i;
    srand (time(NULL));
    for (i=0; i<10; i++)
        printf("%d,\t%d\n", i, rand( ));

    return 0;
}
```

A partir de rand( ) podemos generar números aleatorios con otro tipo de distribución de probabilidad. Por ejemplo, el resultado de

$$\frac{(\text{double})\ \text{rand}(\ )}{\text{RAND\_MAX}+1};$$

es un número real (con precisión *double*) uniformemente distribuidos en el intervalo $[0,1)$. Así, para crear una configuración inicial desordenada de celdas en estado 0 o 1 con la misma probabilidad bastará con asignar al estado de cada una de las celdas el resultado de la comparación: $\dfrac{(\text{double})\ \text{rand}(\ )}{\text{RAND\_MAX}+1.0} < 0.5$

Mostrar el resultado para las reglas de evolución 4, 32, 90, 94, 122, 150, 182 considerando los 500 primeros pasos de un ACE compuesto por 500 celdas. Describir el comportamiento observado en cada caso en función de los dos tipos de comportamiento descritos en el texto.

Ejercicio 15.3. A partir de la función rand( ), ¿cómo podemos generar un número entero pseudo-aleatorio uniformemente distribuido dentro de un intervalo $[a,b]$, siendo ambos enteros?

15.3. Propiedades globales de los ACEs

En esta sección vamos a considerar algunas de las propiedades estadísticas más importantes de los ACEs. Como veremos, este análisis conectará con la *mecánica estadística*, la *teoría de sistemas dinámicos* y con la *teoría formal de la computación*. Llamaremos *s* a una configuración particular del autómata –también denominado estado del autómata– en la que cada celda tiene un determinado valor.

15.3.1. Comportamiento caótico (caos) en las reglas complejas

Vamos a comprobar que en la evolución de los ACEs con reglas complejas se manifiestan las principales propiedades de los sistemas caóticos: impredecibilidad, sensibilidad con respecto a las condiciones iniciales y evolución hacia un atractor extraño con estructura fractal.

Comencemos analizando la principal característica del caos: la sensibilidad con respecto a las condiciones iniciales. Para ello necesitamos definir de algún modo la distancia entre dos estados de nuestro autómata en el espacio de fases y así poder estudiar su evolución. Una medida muy conveniente de la distancia en el espacio de configuraciones del AC es la *distancia de Hamming*, definida como la diferencia entre dos configuraciones del autómata s_1 y s_2, esto es, como el resultado de comparar uno a uno los estados de las celdas en ambas configuraciones y contar los casos en los que son diferentes. Matemáticamente se calcula como

$$H(s_1,s_2) = \#_1(s_1 \oplus s_2),$$

donde la función $\#_1$ calcula el número de unos que se obtiene al aplicar la disyunción exclusiva, que ya vimos en la regla 90, entre los estados de cada pareja de celdas. Así, dos

configuraciones iguales tendrán $H=0$, mientras que dos configuraciones completamente diferentes sin ninguna coincidencia tendrán la máxima distancia de Hamming: $H=N$.

Consideremos el caso de dos configuraciones iniciales s_1 y s_2 que sólo difieren en el estado de una única celda, por lo que están separadas por una distancia de Hamming igual a 1: $H_{t=0}(s_1,s_2)=1$. Si ahora dejamos evolucionar cada una de las configuraciones, después de *t* pasos de simulación esta diferencia inicial podrá afectar a los valores de, como mucho, $2t+1$ celdas, puesto que se propaga lateralmente en cada paso de simulación a los siguientes vecinos, de modo que el valor máximo de la distancia se Hamming será $H_t(s_1,s_2)\le 2t+1$. Lo que se observa es lo siguiente:

- En el caso de las reglas simples la diferencia permanece localizada a unas pocas celdas, por lo que la distancia de Hamming entre las dos configuraciones tiende rápidamente con el tiempo hacia un valor constante pequeño.

- Sin embargo, en el caso de las reglas complejas la distancia de Hamming crece con el tiempo de la forma

$$H_t \sim t^{\alpha}, \text{ con } \alpha = 0.59 \text{ o } 1 \text{ dependiendo de la regla},$$

de modo que configuraciones muy próximas acaban divergiendo en el tiempo, como en los sistemas dinámicos caóticos.

Nótese que este análisis es similar al cálculo de los exponentes de Lyapunov en los sistemas dinámicos. En la Fig. 15.8 se muestra como la distancia de Hamming aumenta con el tiempo en el caso de las reglas complejas 90 y 126.

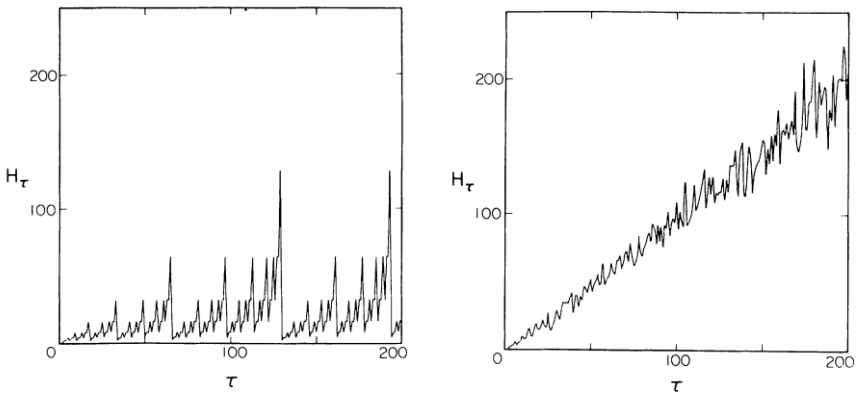


Figura 15.8: Evolución temporal de la distancia de Hamming entre dos configuraciones perfectamente desordenadas que inicialmente diferían únicamente en el valor de una celda. En ambos casos se han utilizado reglas complejas. El resultado de la izquierda corresponde a la regla 90. Se observa que no tiene un comportamiento monótono, es decir, aparecen saltos. Sin embargo, si promediamos la distancia de Hamming sobre muchos pasos temporales para suavizar el comportamiento en el tiempo obtenemos $H_t \sim t^{0.59}$. El resultado de la derecha corresponde a la regla 126 y el comportamiento está mucho más definido; aparte de las pequeñas fluctuaciones parece crecer linealmente con el tiempo: $H_t \sim t^1$. En ambos casos las configuraciones acaban divergiendo.

Ejercicio 15.4. Calcular y representar con Gnuplot la variación de la distancia de Hamming durante la evolución de ACEs con las reglas 4, 32, 90, 94, 122, 150, 182. Para cada regla simularemos dos ACEs de 1000 celdas que comienzan en la misma configuración inicial desordenada (construida aleatoriamente), con la única diferencia de que las celdas del medio tienen diferentes estados: $a'[N/2] = !a[N/2]$.

Nota: Para comparar los estados de las celdas de los dos ACEs en cada paso de tiempo consideraremos que estos estados son elementos booleanos: 0 y 1, y el resultado de la comparación lo obtendremos aplicando el operador lógico XOR a estos dos elementos. Como ya vimos en el Tema 13: Fractales, XOR significa "eXclusive OR", esto es, la disyunción lógica "O pero no Y". La tabla de verdad de esta operación lógica es la siguiente:

A	*B*	A **XOR** B
0	0	0
1	0	1
0	1	1
1	1	0

Como ya dijimos entonces, en C los operadores del álgebra booleana se construyen con un único carácter, así "O" es " | ", "Y" es " & ", y nuestro XOR "O pero no Y" es " ^ ". Estos operadores actúan al nivel de los bits de los números, por eso es tan adecuado su uso cuando los estados pueden ser 0 (todos sus bits son cero) o 1 (todos sus bits son cero menos uno, que es 1).

En este ejercicio consideraremos los 500 primeros pasos de la evolución y exportaremos las parejas de datos $\left(t, H_t(s_1, s_2)\right)$ a un archivo de texto plano que luego representaremos con Gnuplot utilizando el comando:

plot "hamming.dat" with lines

Si suponemos que la evolución temporal de la distancia de Hamming sigue una ley de potencias del tipo $H_t \sim t^{\alpha}$, podemos obtener el exponente α a partir del ajuste por mínimos cuadrados de los logaritmos de los datos a la recta $\log H_t = a + \alpha \log t$. La pendiente de la recta resultante es el exponente de crecimiento de la distancia de Hamming. En Gnuplot se puede obtener esta pendiente a partir de los datos (guardados en un archivo "hamming.dat") mediante el siguiente comando:

fit a+b*x "hamming.dat" u (**log**($1)): (**log**($2)) via a, b

El comando "fit" de Gnuplot proporciona un ajuste por mínimos cuadrados de la expresión a+b*x a los datos (que se leen del archivo y se convierten en sus logaritmos). Para comprobar la bondad del ajuste se puede hacer:

plot a+b*x, "hamming.dat" u (**log**($1)): (**log**($2)) pt 7 ps 0.5

En este ejercicio se mostrará, para cada una de las reglas indicadas más arriba, una gráfica con la evolución temporal de la distancia de Hamming en coordenadas normales y se explicará su comportamiento. Deberá mostrarse también el exponente

obtenido del ajuste con Gnuplot y la figura con los resultados del programa y la recta de ajuste del Gnuplot.

Ejercicio 15.5. Aunque el ajuste por mínimos cuadrados de Gnuplot es muy completo y flexible, nosotros ya tenemos un programa que calcula regresiones lineales (ejercicio de la sección 11.9), y las regresiones a leyes de potencias se pueden convertir, como hemos visto, en simples regresiones a rectas aplicando la función logaritmo. Así, de $H_t \sim t^{\alpha}$ pasamos a

$$\log H_t = a + \alpha \log t$$

Modificar la función de regresión lineal del ejercicio de la sección 11.9 para que ajuste una ley de potencias. Integrar esta función en el programa del ejercicio anterior para que el mismo programa calcule el exponente de Hamming. Compárese el resultado obtenido con este procedimiento con el obtenido con el método de ajuste iterativo de Gnuplot. Para poder comparar el ajuste realizado con nuestro programa con el resultado del Gnuplot debemos utilizar la misma configuración inicial.

A continuación vamos a comprobar que los atractores de las reglas complejas son *extraños* y que tienen una estructura fractal, lo cual es otro indicador de caos.

Si el tamaño del autómata es *N*, cada configuración *s* estará unívocamente descrita mediante un entero binario (compuesto de ceros y unos) de longitud *N* cuyos dígitos son los valores de las celdas del autómata. Por ejemplo, en un ACE compuesto por $N = 10$ celdas, la configuración 1001010111 estará indicada por el entero 599:

$$599 = 1\times 2^0 + 1\times 2^1 + 1\times 2^2 + 0\times 2^3 + 1\times 2^4 + 0\times 2^5 + 1\times 2^6 + 0\times 2^7 + 0\times 2^8 + 1\times 2^9 .$$

De este modo tendremos 2^N posibles configuraciones,[5] por lo que el espacio de todas las posibles configuraciones o *espacio de fases* de nuestros ACEs estará compuesto de 2^N estados: cada estado o configuración *s* corresponde a un punto en el espacio de fases. La evolución del autómata celular lleva a cada configuración inicial a trazar una trayectoria en el tiempo dentro del espacio de las 2^N posibles configuraciones; los puntos de esa trayectoria son las configuraciones por las que pasa el autómata con el tiempo.

En el Listado 15.2 se muestra cómo podemos obtener de forma eficiente todos los posibles estados de un ACE, esto es, su espacio de fases.

Listado 15.2: Programa que calcula y representa todos los estados posibles de un ACE.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define N  5    /* tamaño del ACE */

int main (int argc, char** argv)
{
        int estado, j;
```

[5] Variaciones con repetición de 2 elementos –estados 0 y 1– tomados de *N* en *N*.

```
        for (estado=0; estado < pow (2,N); estado++)
        {
                printf("estado %d: (", estado);
                for ( j=N-1; j>=0; j--)
                {
                        printf("%d ", (estado >> j) & 1);
                }
                printf(")\n");
        }

        return 0;
}
```

Como vamos a ver más adelante, la propiedad de irreversibilidad en la evolución de las reglas de los ACEs hace que las trayectorias en el espacio de fases que comienzan en distintos estados iniciales acaben convergiendo en el tiempo y que, después de muchos pasos temporales, acaben concentradas en atractores. Estos atractores típicamente contienen sólo una muy pequeña fracción de estados posibles. De este modo, la evolución del autómata modifica la probabilidad que tiene cada una de las 2^N posibles configuraciones de ser visitada durante la evolución. En la Fig. 15.9 se muestra la evolución de estas probabilidades durante los 10 primeros pasos de la simulación de tres reglas complejas: 18, 90 y 126, con $N=10$. En cada regla, para calcular estas probabilidades, se han realizado muchas simulaciones partiendo de configuraciones iniciales obtenidas de un conjunto que contiene todas las configuraciones iniciales posibles con igual probabilidad. El eje horizontal representa el espacio de fases del sistema y está compuesto por todos los enteros que van desde 0 a 1023. De este modo, cada punto representa a cada una de las 1024 configuraciones posibles del autómata.

La evolución temporal está dada por las líneas sucesivas, correspondiendo la línea superior al momento inicial $t=0$. En cada paso de tiempo se ha colocado un punto en aquella posición del eje que representa una configuración que ocurre con probabilidad no nula, es decir, que ha sido visitada en ese paso de la evolución por alguna de las simulaciones. Como las configuraciones iniciales para las simulaciones eran tomadas de un conjunto equiprobable de configuraciones que contenía todas las configuraciones posibles, las 1024 son visitadas con igual probabilidad en $t=0$, por lo que en ese tiempo aparece un punto encima de todos los estados posibles: todo el espacio de fases está ocupado. Como se puede comprobar en la figura, la evolución del autómata celular modifica las probabilidades de las diferentes configuraciones, haciendo que la probabilidad de algunas de ellas sea 0, es decir, que nunca sean visitadas en ese paso de la evolución para ninguna condición inicial, de modo que se van formando huecos. Después de unos pocos pasos de evolución se llega a un equilibrio en el que tenemos un conjunto de estados o configuraciones con una probabilidad no nula y que constituyen el *atractor* del sistema. Este atractor tiene la forma de un conjunto de puntos en el eje real distribuidos de un modo fractal, por lo que se trata de un *atractor extraño* (véase el Tema 14. Sistemas dinámicos). En efecto, los puntos que constituyen el atractor extraño de las reglas complejas forman un *conjunto de Cantor*, uno de los fractales más conocidos (véase el Tema 13. Fractales). La dimensionalidad fractal del conjunto de Cantor está dada por la entropía, discutida más adelante. En el caso de la regla 126, la dimensión fractal del conjunto de Cantor es de 0.5.

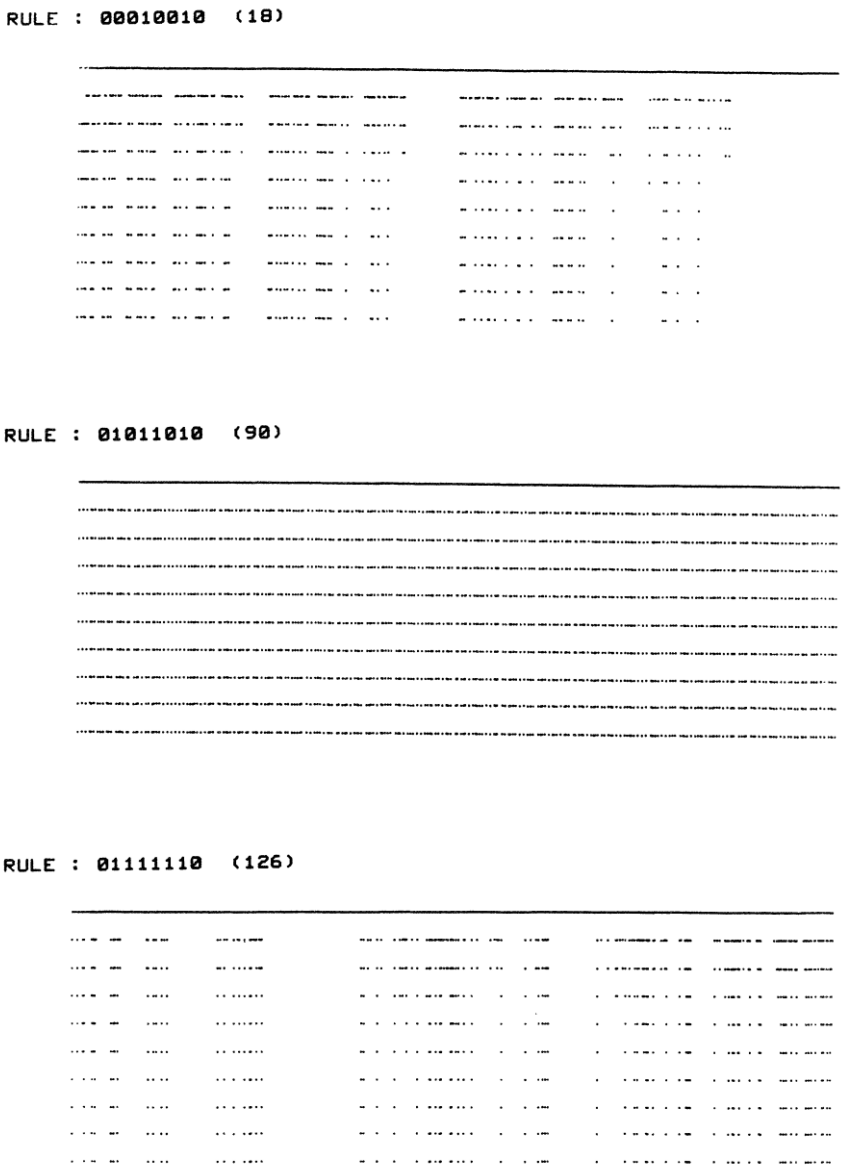


Figura 15.9: Atractores extraños para tres reglas complejas

En el listado 15.3 se muestra el código del programa que exporta los datos necesarios para obtener la Fig. 15.9 correspondiente a cualquier regla. El programa utiliza el código del listado 15.2 para barrer todo el espacio de fases del sistema y así obtener todas las posibles configuraciones iniciales del ACE. Después, para cada una de estas configuraciones o estados iniciales simula la evolución del ACE calculando en cada paso el número de estado en que se encuentra el autómata y guardando en la matriz “prob [t][i]” el número de veces que el estado “i” ha sido visitado en el paso de tiempo “t”. Obsérvese también como al final del programa recorremos esta matriz para verificar que el número total de visitas “contadas” en cada paso sea igual al número de simulaciones, esto es, 2^N. Este tipo de pruebas son habituales (y muy recomendables) cuando se simulan procesos computacionalmente.

Listado 15.3. Algoritmo que calcula el número de veces que cada uno de los 2^N estados del espacio de fases de un ACE es visitado a lo largo de su evolución partiendo de un conjunto de estados iniciales que contiene todas las posibles configuraciones.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define REGLA  90
#define N  10     /* número de celdas del ACE */
#define T  20     /* número de pasos de evolución */


int main (int argc, char** argv)
{
        int i, j, nv, estado, num_estado, cont;
        int ACE[T+1][N+2];

        /* ahora definimos la matriz que guardará las veces que
         * se repita cada estado en el tiempo */

        int prob[T+1][(int) pow (2,N)];

        for (i = 0; i <T+1; i++)
        {
                for (j = 0; j <pow(2,N); j++)
                {
                        prob[ i ] [ j ]=0;
                }
        }

        /* el primer bucle debe recorrer el espacio de fases
        * para proporcionar todos los estados iniciales posibles */

        for (estado=0; estado < pow(2,N); estado++)
        {
                num_estado=0;

                /*definimos el estado inicial del ACE*/
                for (j = N-1; j >= 0; j--)
                {
                        ACE[ 0 ][ N-j ]= (estado >> j) & 1;

                        /*calculamos el número de estado*/
                        num_estado+= ACE[ 0 ][ N-j ] * pow(2,j);
                }

                /*añadimos una entrada en el contador del estado*/
                 prob[ 0 ][ num_estado ]++;

                /*actualizamos las condiciones periódicas de contorno */
                ACE[ 0 ][ 0 ]= ACE[ 0 ][ N ];
                ACE[ 0 ][ N+1 ]=ACE[ 0 ][ 1 ];
```

```
            /* simulamos los T primeros pasos de la evolución del ACE */
            for (i = 1; i < T+1; i++)
            {
                    num_estado=0;
                    for  (j = 1; j <N+1; j++)
                    {
                            nv= (ACE[ i-1 ][ j+1 ] | ACE[ i-1 ][ j ]<<1 | ACE[ i-1 ][ j-1 ]<<2);
                            ACE[ i ][ j ]= (REGLA >> nv) & 1;
                            num_estado+= ACE[ i ][ j ] * pow(2,N-j);
                    }

                    prob[ i ][ num_estado ]++;

                    /* actualizamos las condiciones periódicas de contorno */
                    ACE[ i ][ 0 ]=ACE[ i ][ N ];
                    ACE[ i ][ N+1 ]=ACE[ i ][ 1 ];
            }
    }

    /* hacemos una prueba para verificar que lo que hemos hecho
     * está bien comprobando que en cada paso la suma de las
     * visitas a todos los estados del espacio de fases es igual a 2^N */

    for (i = 0; i <T+1; i++)
    {
            cont=0;
            for (j = 0; j < pow(2,N); j++)
            {
                    cont+= prob[ i ][ j ];
            }

            if (cont != pow (2,N))   printf ("ERROR EN EL PASO %d\n", i);
    }


    /* imprimimos los estados que han sido visitados alguna vez */
    for (i = 0; i <T+1; i++)
    {
            for (j = 0; j < pow(2,N); j++)
            {
                    if (prob[ i ][ j ] != 0) printf("%d\t%d\n", j, i);
            }
    }

    return 0;
}
```

Ejercicio 15.6. Mostrar con Gnuplot los resultados generados del programa del listado 15.3 en los 20 primeros pasos de evolución de las reglas 4, 90, 94, 126, 150 y 182, y discutir los resultados obtenidos. Bastará con redirigir la salida de la ejecución hacia un archivo “atractor_REGLA.dat” y luego utilizar en Gnuplot la sintaxis

plot [0:1024] [-1:21] “atractor_REGLA.dat” with points pt 7 ps 0.2.

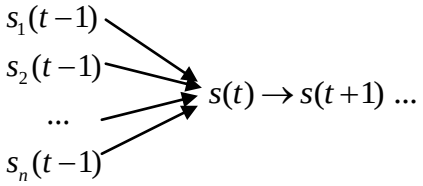
Ejercicio 15.7. El propósito de este ejercicio es representar el número de estados de un ACE visitados en función del tiempo. Para ello modificaremos convenientemente el algoritmo mostrado en el listado 15.3 para que cuente todos los estados que son visitados al cabo de un cierto paso de simulación partiendo de un conjunto de configuraciones iniciales que cubren el espacio de fases. Este número, dividido por el número total de estados posibles 2^N, nos dará la fracción de estados del espacio de fases que son visitados y por tanto el tamaño relativo del atractor. Después representaremos con Gnuplot cómo varía esta fracción relativa con el tiempo (considerar los primeros 20 pasos). Hacer esto para las mismas reglas que en el ejercicio anterior. ¿Qué sucede con la regla 150?

Acabamos de analizar aquellos estados que son visitados al menos una vez en cada paso de tiempo. Sin embargo, dentro de los estados visitados hay estados que son visitados más veces que otros, es decir, tienen más probabilidad de ser visitados por la evolución del autómata si partiéramos de una configuración inicial elegida al azar. Por lo tanto, los estados del atractor tienen diferentes probabilidades entre sí. Las propiedades de aquellas configuraciones más probables dominan las propiedades estadísticas sobre el conjunto, dando lugar a los patrones observados en las configuraciones de equilibrio, como las estructuras en forma de triángulos. De hecho, no es una coincidencia que las reglas que dan lugar a un atractor más “discreto”, con menos estados, como la 126, muestren un patrón de triángulos más claro que las reglas con una distribución más uniforme de estados como la 90.

Ejercicio 15.8. El propósito de este ejercicio es mostrar la distribución de la probabilidad de visita de los diferentes estados del espacio de fases al cabo de 10 pasos para las mismas reglas de evolución que en los dos ejercicios anteriores. De nuevo partimos del programa indicado en el listado 15.3 y lo modificaremos para que imprima la probabilidad de visita de cada estado al cabo de 10 pasos de simulación. Esta probabilidad de visita viene dada por el número de visitas a ese estado divido por el número total de visitas. En la figura obtenida con Gnuplot el eje de abcisas debe corresponder al diagrama de fases del sistema (desde 0 a 1023) y el eje de ordenadas debe representar la probabilidad de visita ¿Qué valor debe tener la suma de las probabilidades de visita sobre todos los estados del espacio de fases? ¿Qué sucede con la regla 150?

15.3.2. Irreversibilidad, auto-organización y entropía

Otra característica importante de la evolución de los ACEs es su irreversibilidad. Los conceptos de *reversibilidad* e *irreversibilidad* son cruciales en *Termodinámica* y *Mecánica Estadística*. Las reglas de evolución hacen que diferentes estados del autómata evolucionen hacia el mismo estado, como se indica en el siguiente esquema:



Evolución irreversible

Por esta razón el proceso es irreversible, no podemos recorrer el camino hacia atrás para recuperar la condición inicial. Como acabamos de ver en la sección anterior, esto hace que el número de posibles estados visitados por del autómata disminuya con el tiempo: el número de estados "no visitados" aumenta con la evolución del sistema. En otras palabras, no todas las 2^N posibles configuraciones son equiprobables; la evolución irreversible del autómata selecciona unos estados entre el colectivo total de estados. Todo lo contrario de los procesos reversibles, en los que las trayectorias que representan la evolución temporal de estados diferentes no se cruzan:

$$...s(t-1) \rightarrow s(t) \rightarrow s(t+1) \ ...$$

Evolución reversible

En este caso, el número de posibles configuraciones debe permanecer constante en el tiempo (teorema de Liouville).

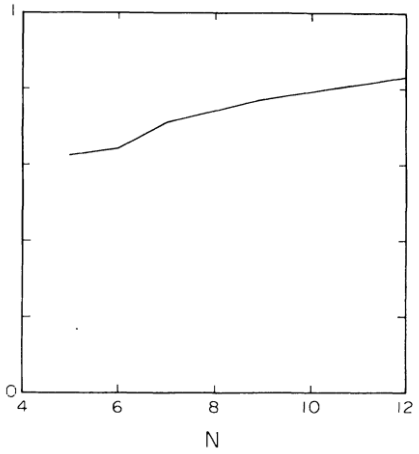


Figura 15.10: Fracción de las 2^N configuraciones posibles que puede tener un ACE de tamaño *N* que no son visitadas por ninguna condición inicial en el primer paso de evolución de acuerdo con la regla compleja 126.

En la sección anterior vimos que la fracción de estados visitados suele disminuir con el tiempo. Lo mismo ocurre con el tamaño del sistema. En la Fig. 15.10 se muestra como aumenta, con el tamaño del sistema, la fracción de configuraciones "prohibidas" (es decir, no visitadas) en el primer paso de la simulación de un ACE utilizando la regla 126. Vemos que esta fracción parece crecer monótonamente hacia 1 a medida que N crece. Cuando *N* es muy grande, sólo una fracción despreciable de todas las posibles configuraciones del ACE es visitada después de unos pocos pasos. La existencia de configuraciones inalcanzables es una consecuencia de la irreversibilidad de la evolución de los ACEs.

Ejercicio 15.9. En el Ejercicio 15.7 obtuvimos la evolución temporal de la fracción de estados visitados. Vimos que en la mayoría de los casos esta fracción disminuía con el tiempo tendiendo hacia un valor constante. En este ejercicio modificaremos el listado obtenido en ese ejercicio (el cual a su vez estaba basado en el listado 15.3) para que proporcione la variación de la fracción de estados no visitados con el tamaño *N* del sistema. El programa debe recorrer un rango de tamaños que van desde $N_{\min} = 3$ hasta $N_{\max} = 18$ por ejemplo, aunque este valor máximo está limitado por la memoria, o mejor dicho, por la habilidad que tengamos para gestionar esa memoria, así que se trata de "llegar hasta donde podamos". Para cada tamaño se

simulará la evolución del ACE, partiendo como siempre de un conjunto de condiciones iniciales dado por el espacio de fases del sistema, hasta un cierto tiempo en el que se calculará la fracción de estados no visitados por el ACE. Si queremos representar el valor constante al que llega la evolución de esta fracción con el tiempo podemos elegir un tiempo suficientemente alto (20 pasos serán suficientes). Obsérvese que ya no necesitamos almacenar la distribución de los estados visitados en cada paso de tiempo (sólo nos interesa el último paso de evolución), por lo que la matriz "prob[T+1][2^N]" puede ser sustituida por un vector "prob[2^N]". Así ahorraremos "memoria" y podremos llegar a mayores valores de *N*. Una buena forma de comprobar que nuestro programa funciona correctamente es reproducir la Fig. 15.10, obtenida después del primer paso. Representar con Gnuplot el resultado obtenido para las reglas 4, 90, 94, 126, 150 y 182. ¿Es siempre irreversible la evolución de los ACEs?

Este proceso de "selección" de estados a partir de configuraciones iniciales arbitrarias puede ser interpretado como un proceso de *auto-organización*, entendido como el proceso por el que un cierto grado de orden o estructuración se desarrolla espontáneamente con el tiempo a partir de un estado inicial sin estructura (véanse las estructuras triangulares observadas en numerosas reglas complejas). La auto-organización en el tiempo se manifiesta en una disminución de la *entropía* del sistema. La entropía es una medida del desorden de un sistema y puede calcularse de la siguiente forma

$$S(t) = -\frac{1}{N}\sum_i p_i(t)\log_2 p_i(t)$$

donde $p_i(t)$ es la probabilidad de que el estado s_i sea visitado en el tiempo *t* de simulación y el sumatorio se extiende sobre las 2^N configuraciones posibles. De este modo la entropía *S* proporciona una medida del número promedio de estados posibles del sistema. La entropía es máxima cuando el sistema está completamente desorganizado. Esto ocurre por ejemplo cuando las 2^N configuraciones son equiprobables, es decir, cuando tienen la misma probabilidad de ocurrir. Entonces tenemos que $p_i = 2^{-N}\ \forall i$ (fíjese que siempre debe cumplirse que $\sum_i p_i = 1$) de modo que

$$S_{\max}(t) = -\frac{1}{N}\sum_i p_i(t)\log_2 p_i(t) = -\frac{1}{N}2^N 2^{-N}\log_2 2^{-N} = 1$$

En la Fig. 15.11 se muestra cómo la entropía disminuye con el tiempo para la regla compleja 126 hasta alcanzar un valor de equilibrio constante. De nuevo, para obtener $p_i(t)$ se ha utilizado el conjunto de estados iniciales dado por el espacio de fases del ACE.

Ejercicio 15.10. Utilizando los programas anteriores obtener y representar con Gnuplot la variación temporal de la entropía para las reglas 4, 90, 94, 126, 150 y 182 en un ACE con $N = 10$. De modo similar al Ejercicio 15.9, representar la variación del valor estacionario de la entropía con el tamaño del sistema. En algunos casos aparecen entropías nulas, ¿podría explicar a qué se debe este resultado?

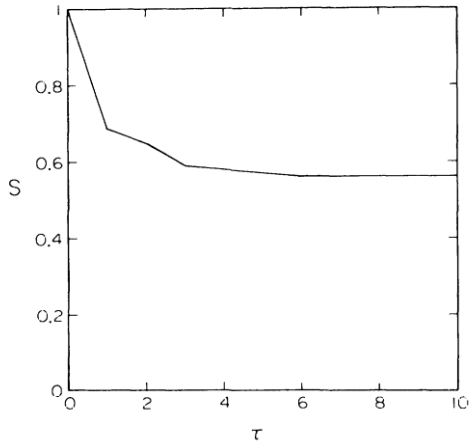


Figura 15.11: Evolución temporal de la entropía para un ACE con $N=10$ evolucionando de acuerdo con la regla 126 a partir de un conjunto de estados iniciales dado por el espacio de fases.

15.3.3 Universalidades

Hasta ahora hemos restringido nuestro análisis a ACEs en los que cada celda tenía dos estados posibles ($k=2$), y una vecindad que abarcaba los primeros vecinos ($r=1$) (*nearest neighbors*), de modo que el estado de la celda es una función del estado de las $2r+1$ celdas de su vecindad en el paso anterior. Sin embargo podemos considerar otros ACES en los que $k>2$ y $r>1$.

Ejercicio 15.11. Deducir la expresión del número total de reglas posibles de un ACE con k estados por celda y una vecindad que llega hasta los *r* vecinos. ¿Cómo podremos expresar cada regla?

Ejercicio 15.12. Modificar el listado 15.1 para que simule la evolución de un ACE con *k* arbitrario y $r=1$ a partir de una semilla inicial con la regla "módulo-*k*". Recordamos que la regla módulo-*k* se aplica sumando los estados de los dos vecinos laterales (no se considera el estado anterior de la propia celda) y devuelve el resto de la división entera de esta suma entre *k*. A esta operación se la denomina suma módulo *k* y devuelve un entero entre 0 y $k-1$. La expresión en C que realiza la operación módulo *k* de un entero a es $a\%k$. Representar la imagen generada en los casos $k=10$ y $k=20$ considerando una semilla inicial con estado 1.

A medida que aumenta *k* o *r* el número de reglas posibles aumenta muy rápidamente y su representación se hace "inmanejable". Sin embargo, podemos reducir el conjunto de reglas agrupándolas en códigos. Por ejemplo, podemos definir un cierto tipo de reglas, denominadas "totalísticas", cuyo resultado final es una función de la suma de los estados de las celdas pertenecientes a la vecindad, es decir:

$$a_i[t] = \mathbf{f}\left[\sum_{j=-r}^{r} a_j[t-1]\right]$$

Esto quiere decir que el estado de una celda dependerá únicamente de la suma de los estados de sus celdas vecinas independientemente de cómo estén distribuidos en la vecindad (de esta forma estamos agrupando distintas reglas bajo un mismo código). Por ejemplo, el resultado de las vecindades (1,1,0), (1,0,1), (0,1,1) en los ACEs que hemos estado viendo será el mismo, y a ello se le asignará un código con un valor numérico (del mismo modo que a cada regla se la asignaba un número, en formato binario o en formato decimal). Para representar numéricamente ese código debemos darnos cuenta que el sumatorio de los estados de una vecindad es un número que va desde 0 hasta $(2r+1)(k-1)$. Este valor máximo se obtiene cuando todas las celdas de la vecindad están en el estado máximo $k-1$. Para cada uno de estos valores nuestro código debe devolver el nuevo estado de la celda entre 0 y $k-1$, por consiguiente, cada código vendrá representado por una sucesión de $(2r+1)(k-1)+1$ números cuyos valores variarán entre 0 y $k-1$. Esto será equivalente a un número en base-*k* cuya expresión decimal será el número del código, **C,** y que se calculará como

$$\mathbf{C} = \sum_{j=0}^{(2r+1)(k-1)} k^j \mathbf{f}[j]$$

Por ejemplo, supongamos un ACE con $k=2$ y $r=2$. En este caso la suma total de los estados sobre la vecindad es un entero que irá desde 0 hasta 5, por lo que nuestro código estará indicado por una secuencia binaria de 6 dígitos. Así, el código **C**=101110, también denominado **Código** 46, devolverá un 1 cuando la suma sea 5, un 0 cuando la suma sea 4, un 1 si es 3, un 1 si vale 2, un 1 si es 1 y un 0 cuando la suma es 0. Si tenemos un ACE con $k=3$ y $r=1$, el código constará de 7 dígitos que representarán un número en base-3. Por ejemplo, al código **C**=2101210 le corresponde el **Código** 1749.

Ejercicio 15.13. Modificar el listado obtenido en el Ejercicio 15.2 para que simule la evolución de un ACE con $k=2$ y $r=2$ (vecindad hasta los *next-nearest neighbors*) a partir de una configuración inicial completamente desordenada. Representar el patrón obtenido utilizando los **Códigos** 10, 20, 50 y 52. **Atención** con las condiciones periódicas de contorno.

El estudio de los ACs elementales nos revela uno de los objetivos má**s** importantes de la física estadística, la búsqueda de *universalidades*. Estas universalidades nos permiten clasificar todos los posibles tipos de comportamiento de un mismo sistema en unos pocos grupos. Las universalidades están caracterizadas por ecuaciones, exponentes, parámetros,… de forma que dos procesos (dos reglas de evolución en nuestros ACEs, por ejemplo) que pertenezcan a la misma universalidad, exhibirán el mismo comportamiento cualitativo y tendrán las mismas propiedades estadísticas, aunque en detalle serán diferentes.

En nuestro caso, hemos visto que las propiedades estadísticas de las estructuras generadas por los autómatas celulares elementales con $k=2$ y $r=1$ caen en dos clases de universalidad, independientemente de los detalles del estado inicial o de las reglas de evolución. Es lógico pensar que esta descripción debe complicarse a medida que

aumentamos el número de estados posibles de una celda o extendemos la vecindad espacial a más vecinos (como acabamos de hacer en los ejercicios anteriores). Sin embargo, es sorprendente saber que, aun considerando todas las posibilidades, podemos reducir todos los tipos de comportamiento a cuatro tipos de universalidad, con propiedades estadísticas bien definidas, que se distinguen sobre todo por los diferentes niveles de predicibilidad del estado final a partir de una configuración inicial de partida conocida. En la Fig. 15.12 se puede ver un ejemplo de cada clase de universalidad.

- Clase 1. Todas las configuraciones iniciales evolucionan después de un número finito de pasos temporales hacia una única configuración final que además es homogénea: todas las celdas tienen el mismo valor (Fig. 15.12 arriba izquierda). Se puede considerar que estos ACEs evolucionan en el espacio de fases hacia un punto límite destruyendo toda la información contenida en el estado inicial. Las dimensiones espaciales y temporales de estos atractores son cero. La predecibilidad es completa: la configuración final del autómata después de la simulación está determinada con probabilidad 1 independientemente de las condiciones iniciales.

- Clase 2. En este caso, la evolución del ACE conduce hacia un estado final estacionario no homogéneo, es decir, hacia un punto límite en el que las celdas no están en el mismo estado, o hacia una estructura periódica que se repite en el tiempo, esto es, un ciclo límite (véase la introducción del Tema 14. Sistemas dinámicos). Esto se manifiesta gráficamente en forma de un conjunto de estructuras simples que están separadas, y que pueden ser estables (no evolucionan en el tiempo) o periódicas (típicamente con periodos pequeños). Véase por ejemplo Fig. 15.12 arriba derecha. En este caso, el estado de una celda particular al cabo de un cierto tiempo sólo depende del valor de los estados de las celdas en una región limitada del estado inicial.

- Clase 3. La evolución conduce hacia un patrón caótico o atractor extraño (véase el ejemplo de la Fig. 15.12 abajo izquierda). Después de un número suficiente de pasos temporales, las propiedades estadísticas de los patrones son prácticamente las mismas independientemente de las condiciones iniciales. El estado de una celda concreta en un momento t dado depende de los estados iniciales de un conjunto de celdas, y este número aumenta con t, de modo que si esperamos suficiente tiempo tendremos que el estado de cada celda sólo puede ser determinado a partir del conocimiento exacto de la condición inicial, lo cual es un distintivo de los sistemas caóticos (véase la introducción del Tema 14. Sistemas dinámicos). Por consiguiente, a medida que pasa el tiempo, la influencia del estado inicial de una celda se propaga con el tiempo a más celdas vecinas (la distancia de Hamming aumenta). De este modo, los cambios en los estados iniciales se propagan con una velocidad finita. Cualquier predicción de estado final requiere un completo conocimiento del estado inicial. Sin embargo, se conjetura que, dado el número necesario de valores iniciales, el valor de una celda puede ser determinado mediante un algoritmo más simple que el propia ACE.

- Clase 4: La evolución conduce a estructuras complejas localizadas que en las mayorías de los casos evolucionan irregularmente durante varios pasos hasta morir (véase por ejemplo Fig. 15.12 abajo derecha). Sólo en unos pocos casos se forman estructuras estables o periódicas que se mantienen durante toda la evolución (estructuras persistentes), mientras que en otros casos las estructuras pueden propagarse. En la Fig. 15.13 se muestran algunas de las estructuras persistentes generadas por esta clase de ACEs. En esta clase de autómatas, el valor de una celda concreta en un instante de la evolución puede depender de los valores iniciales de muchas celdas. Sin embargo, este valor sólo puede ser determinado mediante un algoritmo equivalente en complejidad a la propia simulación del ACE. Eso quiere decir que la impredecibilidad es máxima, que no podemos hacer ninguna predicción efectiva, y que su comportamiento sólo puede ser determinado mediante la propia simulación. Se dice que las propiedades del comportamiento temporal en el infinito de los

ACEs de la clase 4 son *indecidibles*[6]. Se ha conjeturado que esta clase de ACEs son capaces de la denominada *computación universal*. Esto quiere decir que los autómatas celulares pueden ser vistos como computadores en los que los datos están representados por las configuraciones iniciales y la evolución equivale al procesamiento de los datos. La computación universal implica que condiciones iniciales apropiadas pueden especificar cualquier procedimiento algorítmico finito, es decir, el AC puede servir como un computador capaz de evaluar cualquier función computable por complicada que sea.

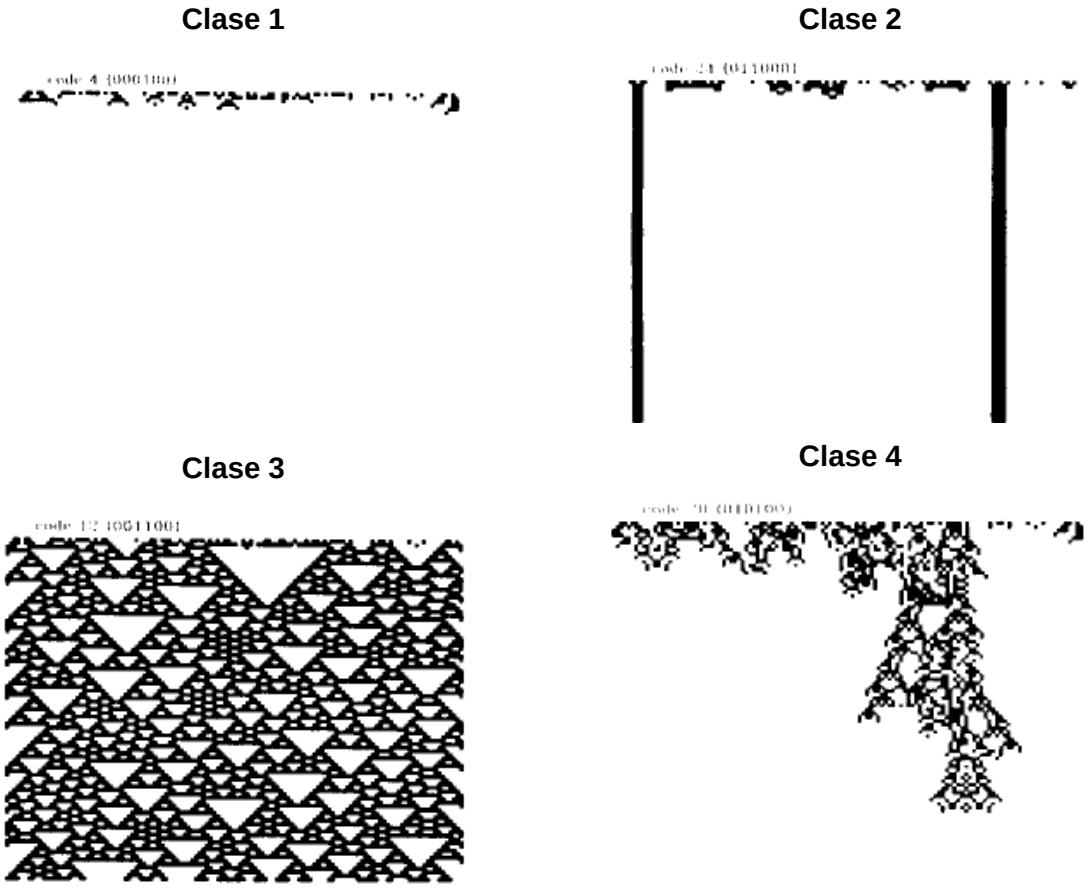


Figura 15.12: Patrones generados durante la evolución de ACEs a partir de configuraciones iniciales desordenadas de acuerdo a reglas pertenecientes a los cuatro tipos de clases de universalidad. En estos ACEs la vecindad se extiende a los segundos vecinos ($k=2, r=2$)

[6] En teoría de la computación, un problema *indecidible* es un tipo de *problema de decisión* para el que es imposible diseñar un algoritmo más simple que dé siempre la respuesta correcta. Se entiende como un *problema de decisión* aquel que tiene como únicas respuestas "*si*" ó "*no*" (1 ó 0) en función de unos valores de entrada. Por ejemplo: dados dos números *x* e *y*, ¿es *x* mayor que *y*? La respuesta a este problema de decisión será "*si*" ó "*no*" dependiendo de los valores que demos a los números *x*, *y*.

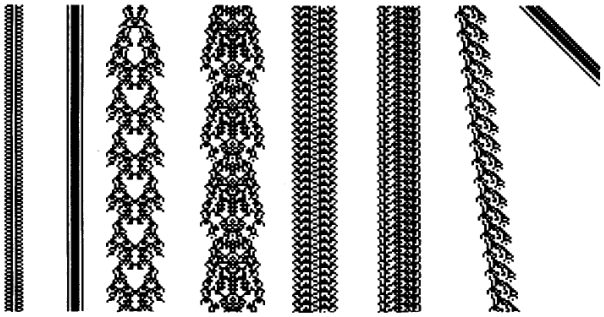


Figura 15.13: Estructuras persistentes observadas durante la evolución de ACEs pertenecientes a la clase 4 con ($k=2, r=2$)

15.4. Autómatas celulares elementales en la naturaleza

Los autómatas celulares han demostrado ser una herramienta muy útil para simular sistemas biológicos, y ello es por dos razones fundamentales. En primer lugar porque los sistemas biológicos están compuestos de células discretas, de modo que la discretización del espacio parece ser una estrategia bastante realista en estos casos. En segundo, porque las interacciones de las células pueden ser fácilmente implementadas en forma de reglas de evolución. A continuación vamos a mostrar como algunos organismos vivos usan de forma natural autómatas celulares elementales en su funcionamiento. El ejemplo típico de ello son los patrones de pigmentación observados en las conchas de moluscos, como por ejemplos las del género *Conus* y *Cymbiola*. En la Fig. 15.14 se muestra el patrón observado en una concha perteneciente al género *Conos Textile* (izquierda) y se compara con el patrón obtenido de un autómata celular elemental (derecha).

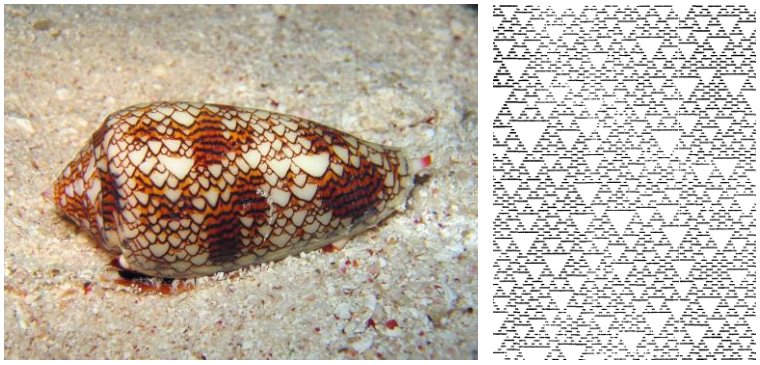


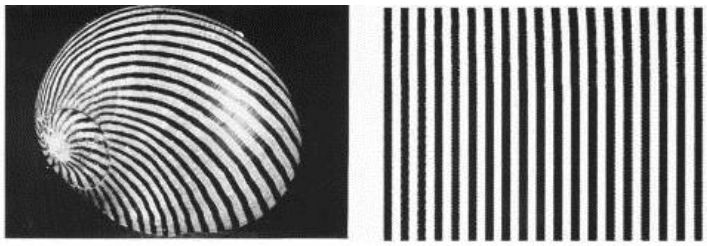
Figura 15.14: (Izquierda) Fotografía de una concha perteneciente al género *Conus textile*. (Derecha) Patrón generado con un autómata celular elemental evolucionando de acuerdo con la regla 126.

Los cromatóforos (células con pigmento) residen en una banda muy estrecha que se extiende a lo largo del labio de la concha. Cada celda segrega pigmento de acuerdo con la señal activadora o inhibidora de sus cromatóforos vecinos, obedeciendo de esta forma natural una regla matemática. La banda de celdas va dejando el patrón coloreado sobre la concha a medida que ésta crece lentamente.

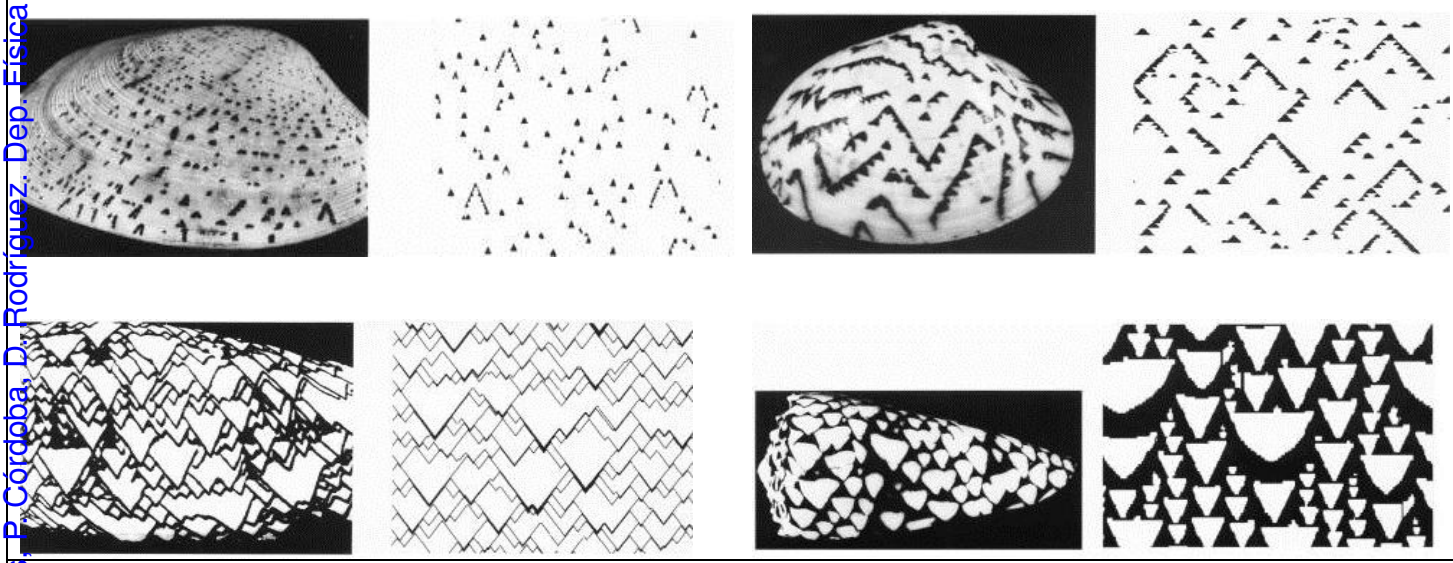
En efecto, como se muestra en la Fig. 15.15, independientemente de la complejidad de la estructura, todas las conchas investigadas parecen poder ser simuladas mediante

autómatas celulares elementales, incluyendo reglas pertenecientes a la "clase 4". Un resultado sorprendente.

Clase 2 (periodicidad)



Clase 3 (caos)



Clase 4 (indecidibilidad)

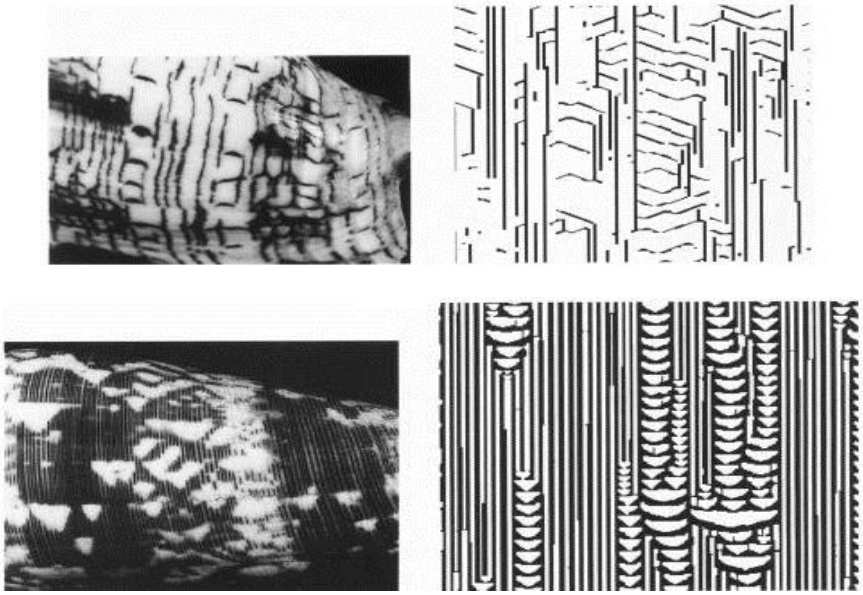


Figura 15.15: (Izquierda) Fotografía de conchas. (Derecha) Simulaciones con autómatas celulares elementales.

15.5. Referencias y bibliografía

Para profundizar en la mecánica estadística de los autómatas celulares elementales proponemos la lectura de los artículos:

- Stephen Wolfram, “Statistical mechanics of cellular automata”, *Reviews of Modern Physics* **55**, 601 (1983).

- Stephen Wolfram, “Universality and complexity in cellular automata”, *Physica D: Nonlinear Phenomena* **10**, 1 (1984).

Estos artículos pueden ser bajados desde la website de Stephen Wolfram:

http://www.stephenwolfram.com/publications/articles/ca/

considerado como uno de los padres de los autómatas celulares. En esa website se pueden encontrar otros artículos relacionados sobre autómatas celulares. Existen también numerosas websites con información, ejemplos y animaciones interactivas sobre autómatas celulares. Recomendamos navegar por el mundo matemático de Wolfram en:

http://mathworld.wolfram.com/topics/CellularAutomata.html