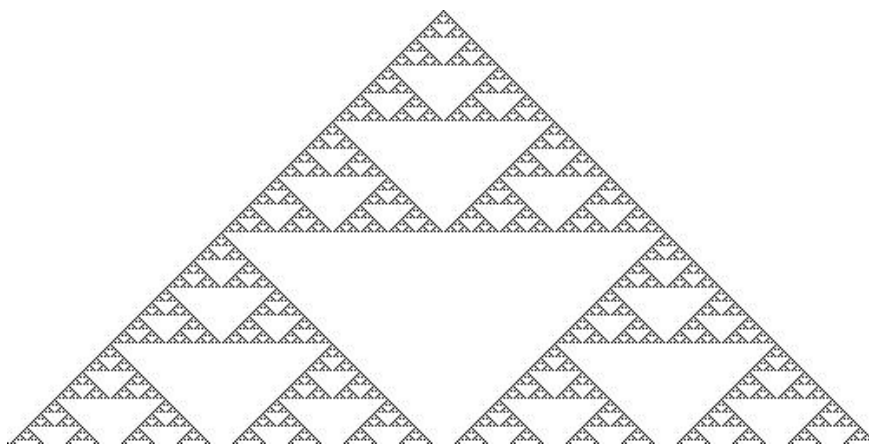


UNIVERSIDAD DE SEVILLA

FACULTAD DE FÍSICA



# **Introducción a los Autómatas Celulares. Enlace como Sistema Complejo**



Autor:

**Daniel López Coto**

Director:

**Dr. Francisco Jiménez Morales**

Curso:

**2015-2016**



# Agradecimientos

En primer lugar me gustaría agradecer al **Dr. Francisco Jiménez Morales** la oportunidad que me ha brindado al poder realizar este trabajo con él. Me ha abierto un nuevo mundo totalmente desconocido para mi en el campo de la física, y ha hecho que me dé cuenta de que pese a las dificultades que te puedas encontrar en el camino, siempre habrá alguien que estará encantado de tenderte una mano amiga.

En segundo lugar, querría darle las gracias a aquella parte del equipo docente que ha sido capaz de transmitirnos su pasión por la física a lo largo de los años con total profesionalidad, y que gracias a ellos amamos esta bella ciencia, que tiene gran parte de arte, que es la física.

En tercer lugar, agradecerle a todos esos compañeros con los que he realizado este viaje, con los cuales he podido forjar algo más que una amistad, pues además, compartimos una misma pasión, y quienes han hecho más llevadero este trayecto.

En último lugar y más importante, quiero darle las gracias a **mi familia**. A **mis padres**, porque sin su ayuda nada de esto sería posible, gracias por vuestra ayuda y apoyo incondicional, y gracias por esos momentos en los que he estado insoportable debido a la presión. A **mi hermana**, pues su gran fuerza de voluntad y empeño constantes han sido un orgullo y objetivo constantes para mi. Y por último, a **mi hermano**, quien me ha servido de inspiración y motivación constante, y quien ha sido hasta día de hoy una referencia y un ejemplo a seguir.



Esta obra está licenciada bajo la Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional.<sup>1</sup>

---

<sup>1</sup>Para ver una copia de esta licencia, visita: <http://creativecommons.org/licenses/by-nc-sa/4.0/>



---

# Índice general

<b>1. Sistemas Complejos</b>	<b>7</b>
1.1. ¿Qué es un sistema complejo? . . . . .	7
1.1.1. Propiedades de los sistemas complejos . . . . .	8
1.2. Algo de historia sobre los sistemas complejos . . . . .	9
1.3. Importancia del estudio de los sistemas complejos . . . . .	10
<b>2. Autómatas Celulares (AC)</b>	<b>11</b>
2.1. ¿Qué son los ACs? . . . . .	11
2.1.1. Los AC como Sistemas Dinámicos . . . . .	14
2.1.2. Algoritmos Genéticos (AG) . . . . .	15
2.2. Algo de historia sobre los ACs . . . . .	17
2.3. Importancia de los ACs . . . . .	18
<b>3. Computación Emergente</b>	<b>19</b>
3.1. Introducción . . . . .	19
3.2. Objetivo . . . . .	20
3.3. Método . . . . .	21
3.4. Resultados . . . . .	24
3.4.1. Función de ajuste y “performance” . . . . .	24
3.4.2. Diagramas de evolución espacio-tiempo . . . . .	26
3.4.3. Análisis . . . . .	30
3.5. Conclusiones . . . . .	32
<b>A. Códigos</b>	<b>34</b>
A.1. Computación Emergente . . . . .	34
A.2. Test . . . . .	41
A.3. Filtro . . . . .	45



---

# Índice de figuras

1.	Hormigas interaccionando de forma local. . . . .	5
1.1.	Ejemplos de sistemas complejos . . . . .	9
2.1.	ACs uni y bidimensionales . . . . .	12
2.2.	Ejemplos de vecindades unidimensionales. . . . .	12
2.3.	Vecindades bidimensionales más representativas. <b>(a)</b> Vecindad de Von Neumann, consistente en la casilla central y los cuatro primeros vecinos más próximos. <b>(b)</b> Vecindad de Moore, la cual añade a las diagonales a la vecindad de Von Neumann. . . . .	13
2.4.	Ejemplo de evolución de la celda central en un AC unidimensional con $r=1$ . La sucesión binaria 01011010 se corresponde con el número decimal 90 y así denominamos a esta regla R90. . . . .	13
2.5.	Diagramas espacio-tiempo (el espacio va en horizontal y el tiempo en vertical en sentido descendente) generados por diferentes ACs. <b>(a) Clase I:</b> regla 45. Tras un breve transitorio, todas las celdas acaban en el estado 1. <b>(b) Clase II:</b> fig. izquierda - regla 4, fig. derecha - regla 122. A diferencia de la clase I el estado final del AC muestra zonas en el estado 1 y otras en el 0 distribuidos de forma regular. <b>(c) Clase III:</b> regla 110. Observamos un comportamiento caótico desarrollado por dicha regla, habiendo partido de una configuración inicial aleatoria. <b>(d) Clase IV:</b> regla totalística 20. En ambas figuras podemos ver cómo partiendo de dos situaciones iniciales aleatorias, el AC (de $r = 2$ ) evoluciona de forma compleja reproduciendo una estructura determinada. . . . .	15
2.6.	Reglas de “El Juego de la Vida”. . . . .	18
3.1.	Esquema del cruce de las reglas. . . . .	23
3.2.	Funciones de ajuste para las mejores reglas de cada generación frente al número de la generación. . . . .	24
3.3.	Evolución de la mejor regla de la <b>primera generación</b> para la primera tanda. . . . .	26
3.4.	Evolución de la mejor regla de la <b>primera generación</b> para la segunda tanda. . . . .	26
3.5.	Evolución de la mejor regla de la <b>tercera generación</b> para la tanda 1 y la 2. . . . .	27
3.6.	Evolución de la mejor regla de la <b>generación 44</b> para la tanda 1, y de la <b>generación 85</b> para la segunda tanda. . . . .	28
3.7.	Evolución de la mejor regla de la <b>generación 100</b> para tres condiciones iniciales diferentes, de la primera tanda. . . . .	29
3.8.	Evolución de la mejor regla de la <b>generación 100</b> para tres condiciones iniciales diferentes, de la segunda tanda. . . . .	29
3.9.	Interacción entre partículas mensajeras. (a) Tanda 1, (b) Tanda 2. . . . .	30





---

# Prólogo

En la naturaleza existe una gran cantidad de sistemas en los cuales la interacción local simple entre sus individuos da lugar a una coordinación global. Algunos ejemplos son: las colonias de insectos, la interacción neuronal y el sistema inmunológico. En los ejemplos citados, hay un proceso común que ocurre, y es el llamado “computación emergente” (en el caso de procesamiento de información) o, con un carácter más general, la aparición de “efectos emergentes”. Este término se refiere a la aparición de efectos globales en el sistema, los cuales no están presentes de forma explícita en los individuos ni en las interacciones. Esta es una de las propiedades más importantes de lo que se conoce como **sistema complejo**.

Muy superficialmente podríamos decir que un sistema complejo es un sistema con una gran cantidad de partes que interactúan entre sí, generalmente de manera no lineal, y que presentan ciertas características como las mencionadas en el párrafo anterior: interacción local y la aparición de efectos emergentes. Un ejemplo muy recurrido para dar una primera visión sobre los sistemas complejos es el caso de un hormiguero. En un hormiguero hay una gran cantidad de individuos, que son las hormigas, y todas son casi idénticas, por lo que para nuestro ejemplo las supondremos idénticas en su totalidad. Además, poseen una interacción muy local, puesto que se comunican con las que tienen inmediatamente a su alrededor sin saber qué es lo que está haciendo el resto del hormiguero. Pese a su interacción local, el hormiguero consigue tener una coordinación global y realizar otro tipo de actividades difíciles.



**Figura 1:** Hormigas interaccionando de forma local.

Debido a la complejidad de estos sistemas se han desarrollado otro tipo de técnicas para abordarlos, además de la resolución analítica que en muchos casos puede llegar a ser extremadamente complicada. La técnica más poderosa a la hora de resolver este tipo de sistemas es la **simulación por ordenador**, y más actualmente puesto que disponemos de una tecnología muy avanzada que nos permite realizar una gran cantidad de cálculos en poco tiempo. Entre las diferentes técnicas que pueden existir para resolver este tipo de sistemas, escogemos la técnica de simulación de los **autómatas celulares (ACs)**, siendo además el paradigma de este tipo de sistemas debido a la simplicidad de los ACs y la gran variedad de problemas a los que se pueden aplicar.

---

Un autómatas celular, descrito brevemente, se trata de un modelo matemático pensado para ser aplicado a sistemas dinámicos (sistemas que evolucionan con el tiempo) y cuya característica más representativa es que tanto el espacio como el tiempo son discretos. Dicho modelo se utiliza como técnica de simulación, ya que la evolución temporal de los ACs se da en pasos discretos de tiempo como hemos dicho antes. También se emplean para simular sistemas naturales, que se puedan entender como un conjunto de individuos, o de partes, que interactúan entre sí de manera local (véase el ejemplo del hormiguero mencionado arriba). Es por este motivo por el cual hemos escogido esta técnica de simulación para abordar los sistemas complejos.

## Objetivo

Los objetivos que persigue el presente trabajo son los siguientes:

- Dar una introducción a los sistemas complejos y la importancia de su estudio.
- Contar brevemente la historia de los sistemas complejos.
- Explicar qué son los autómatas celulares y la importancia de su aplicación en los sistemas complejos.
- Contar una breve historia sobre los autómatas celulares y mostrar algunos ejemplos de autómatas importantes históricamente.
- Aplicación de los autómatas celulares para reproducir el problema de la computación emergente.

## Resumen

En este prólogo se muestra una breve introducción al problema, así como una visión global del mismo.

En el primer y segundo capítulo vemos un poco más a fondo los sistemas complejos y los ACs, tratando de dar una introducción a este tipo de sistemas y procurando explicar qué son los ACs, además de dar una breve introducción histórica sobre ellos, así como algunos ejemplos de ACs que hayan sido relevantes en la historia.

En el tercer capítulo tratamos el problema de la computación emergente. Mostraremos algún gráfico en el que se ve la evolución de un autómatas en función de una determinada regla, así como el mecanismo por el cual se comunican.

Para terminar, en las últimas páginas encontramos los apéndices, donde encontramos el código que hemos desarrollado para tratar el problema de la computación emergente.

# Capítulo 1

## Sistemas Complejos

En este capítulo veremos una definición de sistema complejo y qué características tienen que tener los sistemas para poder considerarlos complejos. También daremos una pincelada de historia sobre los sistemas complejos y la importancia de su estudio.

### 1.1. ¿Qué es un sistema complejo?

Para empezar, es importante remarcar que existen diferentes definiciones para los sistemas complejos. Puesto que los autores no se ponen de acuerdo, para un físico, un biólogo, un informático o un matemático, la definición de sistema complejo es diferente, por lo tanto, nosotros vamos a centrarnos en la definición que proporciona Sayama en su libro *“Introduction to The Modeling and Analysis of Complex Systems”* [8].

En el artículo de Roberto Poli [11] se hace una distinción entre los sistemas complicados y los sistemas complejos.

Una definición de sistema complicado puede ser: sistema formado por varias partes o elementos entrelazados, pudiendo interactuar entre sí de manera lineal o ramificada. La suma de los comportamientos individuales de cada parte o elemento hacen que el sistema funcione como una unidad. El efecto de estos elementos es secuencial lineal, lo que quiere decir que la acción de uno genera un efecto en el siguiente, y así sucesivamente. Por lo tanto, un fallo en el sistema puede venir dado por un solo elemento del sistema, ya sea porque dicho elemento no funcione correctamente de por sí, o porque no esté colocado en el sitio adecuado. Algunos ejemplos de sistemas complicados pueden ser: el motor de un coche, un televisor o un frigorífico.

Como mencionábamos en el prólogo y como Sayama, un sistema complejo es un sistema con una gran cantidad de partes que interactúan entre sí, generalmente de manera no lineal. Los sistemas complejos pueden generar propiedades nuevas y evolucionar mediante la auto-organización, puesto que no son completamente regulares ni completamente

aleatorios, permitiendo el desarrollo de un comportamiento emergente visible a una escala macroscópica. Dichas propiedades emergentes no pueden explicarse mediante las propiedades de los sistemas aislados. Debido a estos comportamientos emergentes, algunos autores suelen decir que los sistemas complejos evolucionan como si de un sistema vivo se tratase.

Algunos ejemplos de sistemas complejos que encontramos en la naturaleza pueden ser: redes neuronales, el sistema fisiológico de un organismo, el clima global, la economía nacional o internacional, e incluso las relaciones sociales y laborales de los seres humanos, aunque este último ejemplo está englobado en el marco de las redes complejas, tema del cual no hablaremos en esta obra.

Por lo tanto, podemos decir que para que un sistema sea considerado complejo, además de poseer varias partes que interactúan entre sí, es necesario que dichas interacciones posean la propiedad de auto-organización, la cual puede generar un comportamiento emergente totalmente inesperado.

### 1.1.1. Propiedades de los sistemas complejos

A continuación tratamos un poco más en detalle las diferentes propiedades que comparten los sistemas complejos.

#### Modelo basado en agentes

Debido las interacciones que poseen este tipo de sistemas, y de sus inesperados resultados, el tratamiento analítico es prácticamente inviable, de manera que nos vemos obligados a recurrir a técnicas de computación para estudiarlos. Una herramienta computacional fundamental para el estudio de los sistemas complejos son los modelos basados en agentes.

Un agente es una unidad autónoma que toma decisiones por sí misma basándose en la interacción local con los agentes de su entorno (denominados vecindad). Existen varios tipos de vecindad y una de ellas es la que poseen los autómatas celulares (vecindad por cuadrículas N-dimensionales).

#### Interacción (generalmente) no lineal

Otra de las propiedades características de los sistemas complejos es la usual ausencia de linealidad en sus interacciones. Ésto, y el elevado número de interacciones entre los individuos de estos sistemas es lo que hace que el tratamiento matemático pueda llegar a ser muy complicado, y es por lo que se recurren a otros métodos de resolución.

Definimos la linealidad, matemáticamente, como la propiedad de aditividad y homogeneidad:  $f(x + \alpha y) = f(x) + \alpha f(y)$ . Debido a que este tipo de sistemas suele carecer de este comportamiento, debemos fijarnos en las propiedades emergentes.

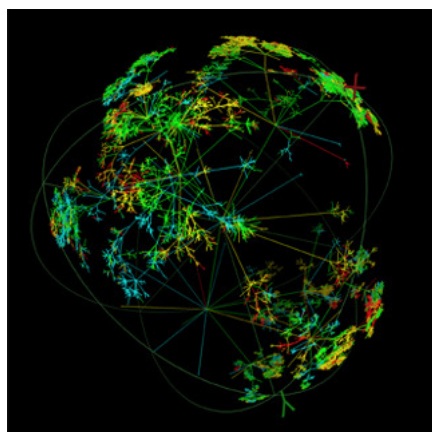
La no linealidad quiere decir que el resultado de las interacciones de estos sistemas no viene dado por una combinación lineal de éstas. Debido a esta propiedad, algunos sistemas complejos derivan en un comportamiento caótico, lo que implica la no existencia de una solución analítica.

### Ausencia de control central

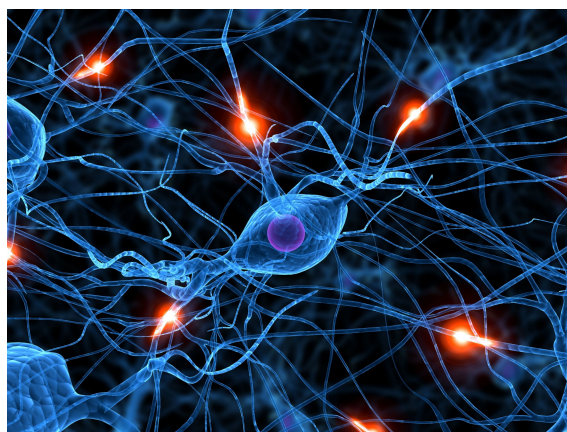
En los sistemas complejos no existe un “cerebro” o mecanismo central que dirija el comportamiento de todo el sistema, cada individuo es totalmente autónomo y realiza su acción en función de como se encuentre su entorno. Debido a esto, surge la propiedad de auto-organización.

### Comportamientos emergentes

Esta propiedad probablemente sea la más importante de los sistemas complejos, o por lo menos la más significativa, puesto que como dijimos antes, debido a la ausencia de linealidad en las interacciones aparecen un conjunto de propiedades emergentes, las cuales no aparecen en los sistemas “complicados”, donde sí está presente la linealidad en sus interacciones.



(a) Internet. Imagen obtenida de: <http://www.ificc.cl/content/introducci%C3%B3n-al-modelamiento-de-sistemas-complejos>



(b) Conexiones Neuronales. Imagen obtenida de: <http://www.elmostrador.cl/vida-en-linea/2016/04/11/fisico-chileno-dictara-curso-introduccion-a-los-sistemas-complejos/>

**Figura 1.1:** Ejemplos de sistemas complejos.

## 1.2. Algo de historia sobre los sistemas complejos

En los años previos al 1900, la física estaba centrada en problemas de dos a cuatro variables aproximadamente, donde era relativamente “fácil” obtener una solución analítica. Estos problemas no representaban las dificultades a las que se tenían que enfrentar las “ciencias de la vida” como son la biología o la medicina, las cuales necesitaban dar respuestas a preguntas como: “¿Cómo aparecen, se organizan y se clasifican efectos concurrentes que aparentemente están correlacionados?”. Dichos problemas no podían ser resueltos por sistemas

**sencillos** como los que se estudiaban hasta la fecha.

No fue hasta después del 1900 cuando científicos como Josiah Willard Gibbs decidieron dar un paso adelante y desarrollar métodos matemáticos y probabilísticos para tratar problemas de millones de variables, los cuales se llamaron problemas de **complejidad desorganizada**.

Lo que quiere decir “problemas de complejidad desorganizada” es que en dichos problemas existen una gran cantidad de variables, en el cual cada una tiene un comportamiento individual totalmente desconocido. Pero a pesar de este comportamiento individual, el sistema posee unas propiedades medias que son ordenadas y analizables.

A estos tipos de problemas se les pueden aplicar los mecanismos de la mecánica estadística para obtener resultados y conclusiones. Algunos ejemplos de este tipo de sistemas puede ser: una compañía de seguros que desconoce la situación concreta de cada asegurado, pero tiene un valor medio de muertes anuales; un conjunto de muchas bolas de billar, en el cual desconoces la posición y velocidad de cada bola individualmente, pero del cual puedes obtener un valor medio de colisiones por minuto, por ejemplo.

A estos dos tipos de problemas se les añade uno nuevo, los conocidos como problemas de **complejidad organizada**. Este tipo de problemas surge debido a que, pese a la potencia del nuevo método para los problemas desorganizados, se dejaba sin estudiar un gran campo, el cual trata las situaciones intermedias. La importancia de estas situaciones no recae en el número de partículas o variables que están envueltas, sino en la forma en la que dichas variables se relacionan.

Estos tres tipos de problemas son los que se definen en el artículo de Warren Weaver, “*Science and Complexity*” [6].

En la actualidad existe una ciencia sobre sistemas complejos centrada en las interconexiones de las partes del sistema, en cómo se organizan y en cómo evolucionan. Es conocida como “Ciencia de las Redes Complejas”.

### 1.3. Importancia del estudio de los sistemas complejos

Como hemos comentado en el apartado anterior, el estudio de este tipo de sistemas estuvo motivado por la necesidad que tenían la biología y de la medicina de responder a ciertas preguntas relevantes y la imposibilidad de hacerlo con los métodos existentes por aquel entonces.

Gracias al estudio de los sistemas complejos podemos conocer los mecanismos por los que se comunican las células (ausencia de control central), cómo se organiza un hormiguero (jerarquía) o cómo, debido al comportamiento de un conjunto de individuos, aparecen ciertos efectos sobre su colectivo (comportamientos emergentes).

## Capítulo 2

# Autómatas Celulares (AC)

En este capítulo hablaremos sobre qué son los AC, daremos una breve introducción histórica sobre ellos y algunos ejemplos de AC. Su relación con los sistemas complejos y la importancia de su uso en dichos sistemas.

### 2.1. ¿Qué son los ACs?

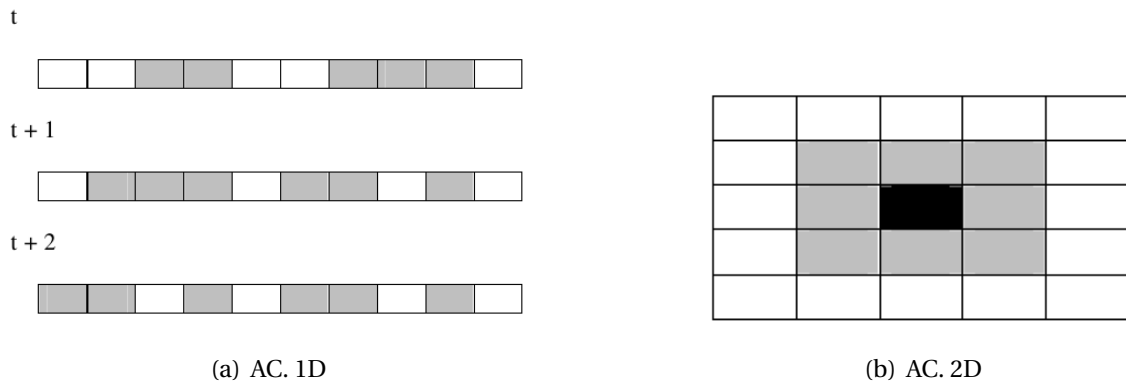
Tal y como se menciona al principio de la obra “Evolución de autómatas celulares utilizando algoritmos genéticos” [7]: *“Los ACs constituyen un modelo matemático de interacción extremadamente simple, discreto en el espacio y en el tiempo, que emula el comportamiento de muchos sistemas naturales”*.

Los ACs están compuestos por un gran número de componentes/individuos/células que se caracterizan por ser homogéneos, estar distribuidos discretamente en el espacio, carecer de control central e interactuar con su vecindad (interacción local).

La evolución de las células de los ACs está definida por un conjunto de reglas preestablecidas, y se realiza en pasos de tiempo discreto. De esta forma, en cada ciclo de tiempo las células evolucionarán según el estado de las células vecinas, evolucionando todas en el mismo instante de tiempo.

Aunque los autómatas que vamos a tratar son los ACs en una dimensión, debido a que se puede hacer una representación espacio-temporal y ver los patrones de comunicación entre las diferentes partes del sistema más fácilmente, existen también autómatas en otras dimensiones.

- **AC unidimensional:** Este caso es el más sencillo de todos, puesto que el autómata consiste en una única fila en la que se disponen las células. Este tipo de autómata es útil cuando queremos ver la evolución del estado de cada célula en función del tiempo.
- **AC bidimensional:** Este tipo de autómata es un tablero en el que cada casilla es una célula. Esta clase de AC presenta diferentes vecindades. La más simple es la vecindad de Neumann, la cual sólo tiene en cuenta las casillas que están inmediatamente arriba, abajo, a la derecha y a la izquierda. Otro tipo de vecindad es la de Moore, la cual añade las diagonales a la vecindad de Neumann.



**Figura 2.1:** La figura (a) representa un AC unidimensional en diferentes estados de tiempo, mostrándose así una evolución. Las zonas sombreadas indican las células que están “activas”, y las blancas las “inactivas”. En la figura (b) se representa un AC bidimensional, en este caso la zona sombreada representa la vecindad, siendo en este caso la de Moore, con un radio de interacción  $r=1$ .

Antes de continuar debemos dar algunas definiciones, las cuales veremos para autómatas unidimensionales pero que se pueden generalizar para más dimensiones:

- **Estado de la celda ( $s_i(t)$ ):** Representa el estado de la celda que ocupa la posición  $i$  en el instante de tiempo  $t$ . El número total de celdas es  $N$  y el conjunto de posibles estados  $\{k\}$ , donde  $s_i(t) \in \{k\}$ . Nosotros vamos a considerar ACs binarios, por lo que  $k = \{0, 1\}$ .
- **Vecindad:**  $V_i(t) = \{s_i(t), s_{i+1}(t), s_{i-1}(t), \dots\}$  indica el conjunto de celdas alrededor de la celda  $i$  y la propia celda  $i$ . Las  $N$  diferentes celdas del AC se disponen en un mallado de dimensión  $d$ . Cuando el mallado tiene un tamaño finito siempre consideraremos condiciones de contorno periódicas, lo que significa que la celda 0 tiene como vecina a la celda  $N-1$  y viceversa. Existen diferentes tipos de vecindades.
  - **Una dimensión ( $d=1$ ):** La vecindad está compuesta por las celdas que están a izquierda y derecha de la celda  $i$  una distancia  $\leq r$  y por la celda  $i$ . El número de vecinos en una dimensión es:  $2r + 1$ , y el número de posibles vecindades  $k^{2r+1}$ .



(a) Vecindad para AC unidimensional con  $r = 1$ .



(b) Vecindad para AC unidimensional con  $r = 2$ .

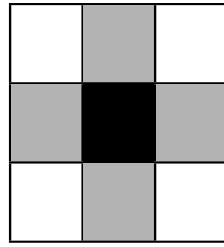
**Figura 2.2:** Ejemplos de vecindades unidimensionales.

- **Dos dimensiones ( $d=2$ ):** En dos dimensiones hay dos vecindades que son las más representativas. La vecindad de Von Neumann y la vecindad de Moore. Figura (2.3).
- **Reglas:**  $R = f(V_i(t))$  Las reglas son un conjunto de funciones que nos darán la evolución temporal de cada celda de nuestro autómata teniendo en cuenta el estado de las celdas de su vecindario en el estado anterior:

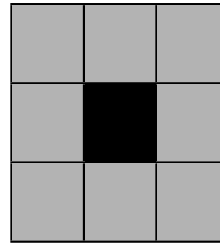
$$s_i(t+1) = f(s_{i-r}(t), \dots, s_{i-1}(t), s_i(t), s_{i+1}(t), \dots, s_{i+r}(t))$$

En la figura (2.4) se muestra un ejemplo.





(a) Von Neumann.



(b) Moore.

**Figura 2.3:** Vecindades bidimensionales más representativas. **(a)** Vecindad de Von Neumann, consistente en la casilla central y los cuatro primeros vecinos más próximos. **(b)** Vecindad de Moore, la cual añade a las diagonales a la vecindad de Von Neumann.

Vecindad en $t$	111	110	101	100	011	010	001	000
Celda central en $t+1$	0	1	0	1	1	0	1	0

**Figura 2.4:** Ejemplo de evolución de la celda central en un AC unidimensional con  $r=1$ . La sucesión binaria 01011010 se corresponde con el número decimal 90 y así denominamos a esta regla R90.

En la figura (2.3) se muestra un ejemplo de vecindad con el valor de salida que tendrá la celda central en el siguiente paso de tiempo según el estado de su vecindad en el paso de tiempo anterior. Debido a la forma en la que aparece la regla: 0-1-0-1-1-0-1-0, podemos codificarla de forma sencilla.

Una forma de codificarla sería la siguiente:

$R = \alpha_7 - \alpha_6 - \alpha_5 - \alpha_4 - \alpha_3 - \alpha_2 - \alpha_1 - \alpha_0$ , donde  $\alpha_i$  puede ser 0 ó 1. De forma que el código numérico de la regla sería:

$$\text{Código } R = 2^0 \cdot \alpha_0 + 2^1 \cdot \alpha_1 + 2^2 \cdot \alpha_2 + 2^3 \cdot \alpha_3 + 2^4 \cdot \alpha_4 + 2^5 \cdot \alpha_5 + 2^6 \cdot \alpha_6 + 2^7 \cdot \alpha_7 = \sum_{i=0}^7 2^i \cdot \alpha_i$$

Pero cuando la regla es bastante más larga, esta codificación resulta incómoda, pues los números que aparecen son demasiado grandes. En estos casos se recurre a convertir la regla a lenguaje hexadecimal.

Existen diferentes **tipos de reglas**:

- **Legales:** son aquellas en que la vecindad nula da siempre un valor nulo ( $\alpha_0 = 0$ ) y además son simétricas, es decir que la vecindad 110 debe dar lo mismo que la 011 ( $\alpha_6 = \alpha_3$ ) y que la 100, lo mismo que la 001 ( $\alpha_4 = \alpha_1$ ).
- **Totalísticas:** aquellas en que la regla de evolución sólo depende de la suma de los estados de los vecinos.  $S(t) = \sum_{j=0}^n s_j(t)$ .
- **Elementales:** son las reglas legales con  $r = 1$  y  $d = 1$ . Sólo hay 32 reglas elementales.

Los estados de los ACs se clasifican generalmente como “activos” e “inactivos”. Los estados inactivos permanecen inactivos si todos sus vecinos también están inactivos. Dichos estados, en muchos modelos, se representan con “0” o con celdas “blancas”. Este estado representa un “vacío” en el espacio de los ACs.

Los estados activos son los que pueden cambiar de forma dinámica e interaccionan con sus vecinos cercanos. Este tipo de estados activos son los que juegan un papel primario en la producción de comportamientos complejos.

Dado que los ACs presentan una extensión espacial, tienen condiciones de contorno además de condiciones iniciales. Hay varios tipos de condiciones de contorno:

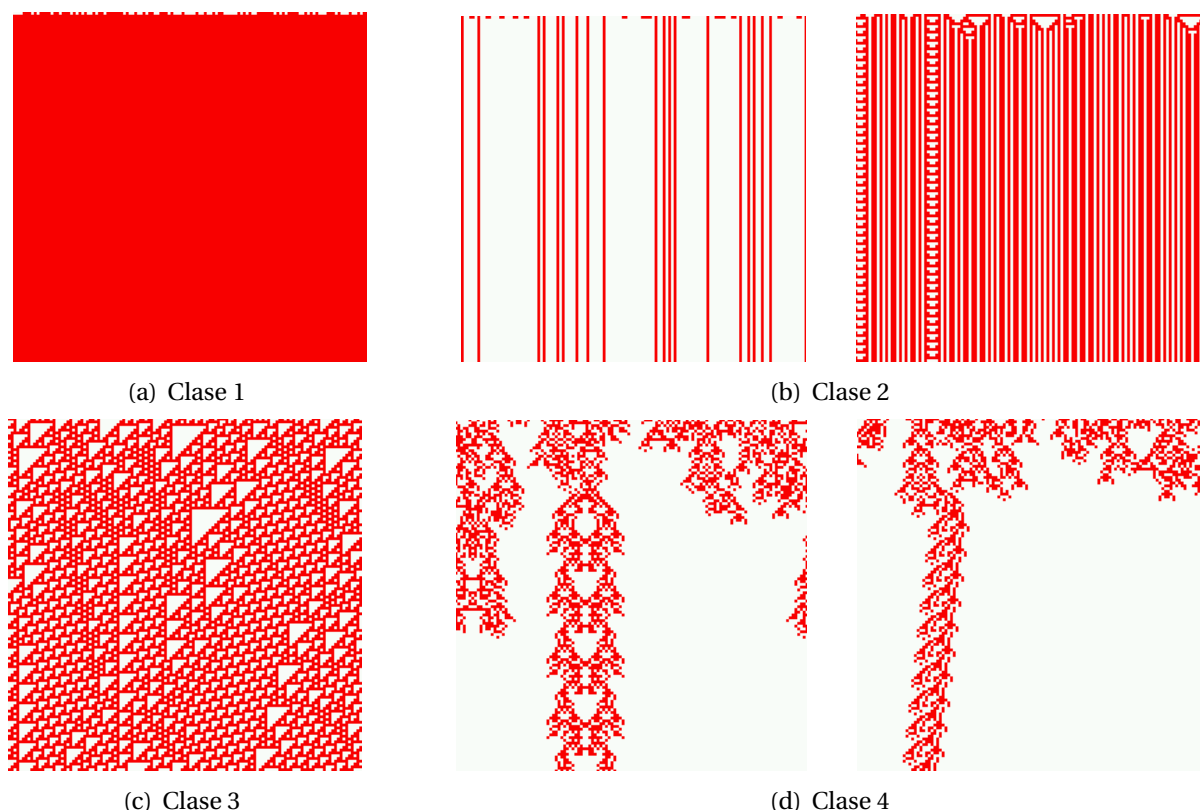
- **Sin contorno:** Se asume que el espacio es infinito lleno completamente de estados “inactivos”.
- **Contorno periódico:** Se asume que el espacio está cerrado “circularmente”, de forma que la celda final está conectada con la del principio.
- **Contorno “aislado”:** Se asume que la primera y última celda de nuestro espacio finito no tiene vecinos tras los límites. En este caso, la función de transición de estados debe ser definida específicamente para que pueda tratar este tipo de caso.
- **Contorno fijo:** Se asume que las celdas primeras y últimas tienen unos vecinos fijos que nunca cambian de estado.

### 2.1.1. Los AC como Sistemas Dinámicos

Según el comportamiento de los autómatas, podemos distinguir diferentes configuraciones que éstos pueden adaptar. Algunos evolucionan hasta que se estabilizan para permanecer inmutados, otros presentan un comportamiento periódico, e incluso algunos muestran comportamientos caóticos.

Debido a esto, Wolfram [Wolfram, 1984a] propuso una clasificación con la cual se podrían identificar la conducta de los ACs. Dichas categorías son:

- **Clase 1:** Todas las configuraciones iniciales convergen tras un periodo transitorio de tiempo a la misma configuración final. Ejemplo: un autómata en el que cualquier configuración inicial terminase en que todas las celdas fuesen unos. Este comportamiento es el equivalente a un punto fijo.
- **Clase 2:** Todas las configuraciones iniciales convergen tras un periodo transitorio a la misma configuración o a algún ciclo periódico de configuraciones.
- **Clase 3:** Algunas configuraciones iniciales convergen hacia una configuración caótica.
- **Clase 4:** Algunas configuraciones iniciales convergen a estructuras complejas que se propagan en el espacio y el tiempo. Pese a que esta clase no está definida de forma exhaustiva, aquí se situarían todos aquellos que fuesen capaces de realizar alguna computación compleja. Los AC de la clase IV no tienen equivalente entre los sistemas dinámicos y han sido objeto de un intenso debate sugiriéndose incluso que la evolución natural daría lugar a este tipo de comportamiento con la famosa hipótesis del “borde del caos” [4].



**Figura 2.5:** Diagramas espacio-tiempo (el espacio va en horizontal y el tiempo en vertical en sentido descendente) generados por diferentes ACs. **(a) Clase I:** regla 45. Tras un breve transitorio, todas las celdas acaban en el estado 1. **(b) Clase II:** fig. izquierda - regla 4, fig. derecha - regla 122. A diferencia de la clase I el estado final del AC muestra zonas en el estado 1 y otras en el 0 distribuidos de forma regular. **(c) Clase III:** regla 110. Observamos un comportamiento caótico desarrollado por dicha regla, habiendo partido de una configuración inicial aleatoria. **(d) Clase IV:** regla totalística 20. En ambas figuras podemos ver cómo partiendo de dos situaciones iniciales aleatorias, el AC (de  $r = 2$ ) evoluciona de forma compleja reproduciendo una estructura determinada.

Los autómatas celulares son modelos matemáticos masivamente paralelos, lo que quiere decir que todas las células evolucionan en el mismo estado de tiempo, de forma paralela, en función de el estado que tenga su vecindad. Debido a esta naturaleza, no existe un control central en los ACs. Pero, debido a su localidad, tienen un mecanismo de comunicación limitado, lo que hace que parezca que poseen mecanismo de auto organización.

### 2.1.2. Algoritmos Genéticos (AG)

En un sentido amplio el estudio de los AC se puede dividir en dos grupos: a) los problemas directos en los cuales se estudian las propiedades y características de una regla dada y b) los problemas indirectos, en los cuales lo que se pretende es encontrar una regla de AC que realice una determinada tarea. Este último tipo de AC es el que va a ser objeto de nuestro estudio.

En el problema que vamos a tratar sobre la computación emergente vamos a necesitar recorrer el espacio de las soluciones para encontrar la que mejor resuelva nuestro problema. Para ello utilizaremos autómatas unidimensionales, con  $k = 2$  y  $r = 3$ , de manera que tendremos una cantidad de reglas enorme ( $N_R = k^{(2r+1)} \approx 3,4 \cdot 10^{38}$ ), por lo que la única forma de poder encontrar las mejores reglas e ir haciéndolas evolucionar es utilizando el método del **algoritmo genético**. Este método no sería necesario en el caso de  $r = 1$ , puesto que podríamos resolver el problema mediante fuerza bruta, algo inconcebible en nuestro caso.

Como dicen Juan Ignacio Vázquez y Javier Oliver en su artículo [7]: *“Los algoritmos genéticos son métodos de búsqueda en el espacio de soluciones de un problema basados en el mecanismo de la selección natural o lucha por la supervivencia”*.

Mediante este método, las soluciones quedan codificadas como si de un gen se tratase, y de la misma forma que se realiza la mezcla de genes en la reproducción sexual, las mejores reglas son escogidas, mantenidas intactas y mezcladas entre sí para generar otras nuevas. Generación tras generación, obtendremos las mejores soluciones, hasta que finalmente, tras un número determinado de generaciones, se pretende tener al menos un individuo que sea capaz de llegar a la solución del problema.

Volvemos a mencionar a Juan Ignacio Vázquez y Javier Oliver en su artículo [7]: *“El algoritmo genético es un algoritmo matemático, altamente paralelo, que transforma un conjunto de objetos (población) representados matemáticamente cada uno de ellos mediante una cadena de símbolos (genotipo) asociada a una determinada medida de adaptación al medio, en otra población distinta (siguiente generación) mediante la aplicación del principio de supervivencia del más fuerte en términos de adaptación.”*

En la siguiente generación entran en juego lo que se llaman “operadores genéticos”:

- **Reproducción:** los mejores individuos son los que son seleccionados para reproducirse, de esta forma se garantiza una descendencia proporcional a esos valores.
- **Mutación:** los genes pueden mutar, de forma que se introducen nuevos individuos en el espacio de soluciones. De esta forma se evita que las soluciones queden encerradas en mínimos locales.
- **Recombinación sexual:** las reglas se parten y se recombinan, generando los descendientes los cuales, generación tras generación, estarán mejor adaptados.

Los autómatas celulares se han descrito como modelos matemáticos masivamente paralelos, extremadamente simples en su concepción, pero capaces de llevar a cabo en algunos casos conductas sumamente complejas y no predecibles a priori, a menos que se efectúe una simulación, debido al enorme número de interacciones entre las células.

Debido a la gran complejidad de las operaciones que pueden llevar a cabo los ACs, es prácticamente imposible diseñar a mano un conjunto de reglas para que un autómata celular realice una tarea compleja. La mayor parte de los descubrimientos son debidos a actos de observación y simulación.

Hemos visto que los AC muestran un amplio rango de comportamientos dinámicos y además son capaces de realizar diferentes clases de computación. De esta forma surgió la propuesta de hacer evolucionar una población de reglas de AC mediante un algoritmo genético para llevar a cabo una tarea que requiera de la cooperación de todas las celdas. De acuerdo con las ideas de Langton [4] se esperaba que un proceso evolutivo como un AG daría lugar a reglas correspondientes a la clase-IV ( la situada entre la zona periódica y la caótica ). Los trabajos iniciales en este campo fueron llevados a cabo por Packard [5] y después continuaron M. Mitchell y J.P.Crutchfield [2] y son parte principal de este trabajo.

## 2.2. Algo de historia sobre los ACs

Se le reconoce a **John von Neumann** la introducción de los ACs al intentar desarrollar un modelo abstracto de auto reproducción en biología. En 1947, Neumann comenzó a pensar en modelos en 3D descritos por ecuaciones en derivadas parciales, pero al poco tiempo cambió de idea. Por la analogía con los esquemas de los circuitos electrónicos se dió cuenta de que un modelo en 2D debería ser suficiente. En 1951 Neumann, sugerido por **Stanislaw Ulam**, simplificó su modelo terminando con un autómata de dos dimensiones. El AC que construyó en 1952-1953 tenía 29 colores posibles por cada célula, y unas reglas complicadas que emulaban específicamente las operaciones de los componentes de un ordenador y varios dispositivos mecánicos.

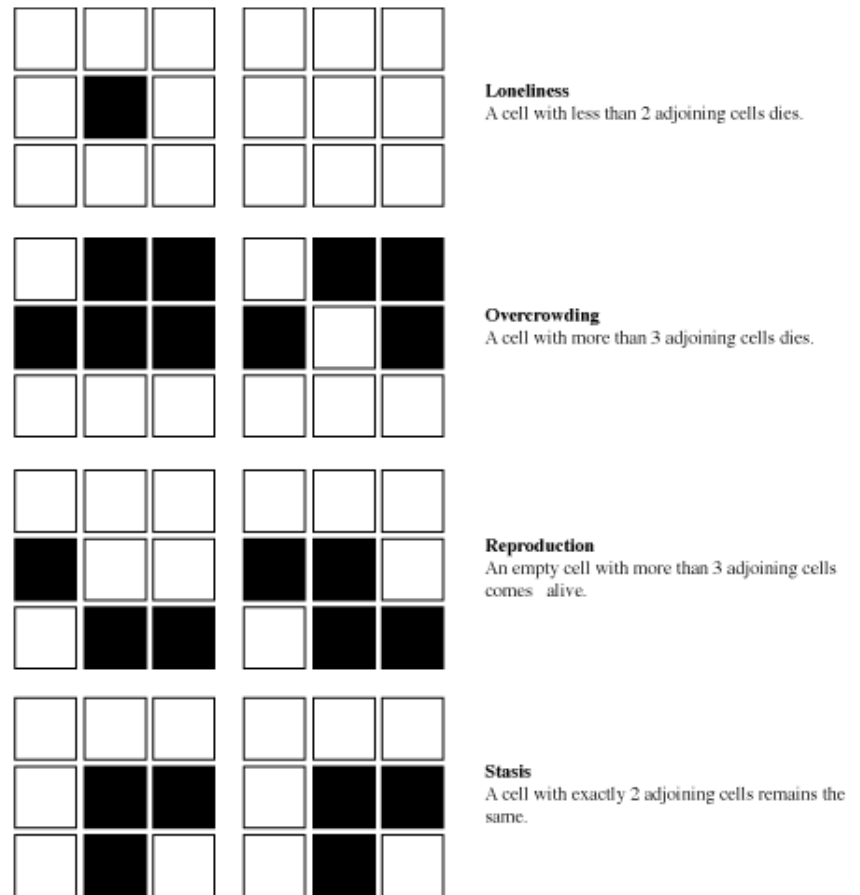
Del trabajo de Neumann surgieron dos nuevas ramas. La primera, aproximadamente en 1960, fue la de construir un AC que se auto reprodujese. La segunda, la de desarrollar y capturar la esencia de la auto reproducción mediante un estudio matemático de las propiedades de los ACs.

Para finales del 1950, se habían dado cuenta que se podían ver los autómatas celulares como computadoras paralelas, y particularmente en el 1960, una serie de aumento de detallados teoremas técnicos, daban prueba formalmente de sus capacidades computacionales. A finales del 1960 comenzaron los esfuerzos por conectar a los autómatas celulares a discusiones matemáticas de sistemas dinámicos.

Uno de los más famosos y estudiados autómatas celulares es el conocido como el “Juego de la Vida” creado por John Conway a finales de los años 60 y popularizado por M. Gardner en la década de los 70. La importancia del “Juego de la Vida” radica en su capacidad de computación universal, desde este punto de vista las condiciones iniciales se interpretan como el input suministrado al AC y la salida es situación final alcanzada por el AC.

Otra importante cuestión acerca de los AC fue la planteada por Toffoli, Margolus y Fredkin [13] en los años 80 en el sentido de si los ACs pudieran modelar las propias leyes de la Física. En este sentido se han formulado diversos modelos computacionales que preservan la información y que en escala microscópica mantienen la reversibilidad como por ejemplo la ecuación de transmisión del calor, la ecuación de ondas y de Navier-Stokes.

Finalmente habría que señalar el sistemático estudio de los AC llevado a cabo por S. Wolfram [9] a finales de los años 80 y su relación con los sistemas dinámicos y que con el advenimiento de los ordenadores hicieron de los AC una de las principales técnicas de simulación en campos de investigación tan dispares como la sociología, la geografía, la biología, la química y la física [1].



**Figura 2.6:** Reglas de “El Juego de la Vida”.

## 2.3. Importancia de los ACs

Pese a la simplicidad de los ACs, éstos son de gran utilidad debido a las propiedades que hemos mencionado en los apartados previos. Gracias a ellos se pueden resolver infinidad de problemas de forma alternativa a la resolución analítica.

Algunos ejemplos de simulaciones en las que se utilizan autómatas celulares son:

- Propagación un incendio o una enfermedad.
- Efectos difusivos.
- Movimientos migratorios de las aves.
- Modelo de segregación [12].

# Capítulo 3

## Computación Emergente

En este capítulo tratamos el problema de la computación emergente. En primer lugar damos una introducción al problema, explicando de qué trata. En segundo lugar exponemos el método seguido para realizar el estudio. Por último lugar mostramos los resultados obtenidos.

### 3.1. Introducción

La **computación emergente** es un término que se emplea de forma general para describir la aparición del procesamiento global de la información en un sistema [2].

La propia naturaleza ha creado sistemas en los que se ponen de manifiesto las habilidades de procesamiento de información que surgen a partir de la interacción local de los individuos que pertenecen a ese sistema. Algunos ejemplos los hemos mencionado antes, como puede ser el caso del hormiguero, donde las hormigas se comunican entre ellas de forma local y a partir de estas interacciones todo el hormiguero procesa la información y evoluciona en consecuencia. Otro ejemplo es el de la red neuronal, en el que cada neurona se comunica con las que tiene a su alrededor, enviando señales eléctricas y procesando parte de la información.

El desarrollo de una coordinación global, que surge a partir del conjunto de componentes simples y descentralizados que son capaces de procesar la información, tiene importantes ventajas frente al procesamiento centralizado de la información, tanto en la naturaleza como en los mecanismos de procesamiento de información que construimos los humanos.

1. **Velocidad.** Un sistema centralizado puede sufrir el efecto de “cuello de botella” al procesar la información, y por lo tanto procesarla más lentamente. Los sistemas descentralizados tienen diferentes “regiones” donde se procesa la información, y al estar repartida ésta, resulta más difícil que sufra dicho efecto.
2. **Robustez.** Si el sistema central sufre daños, todo el sistema colapsa, mientras que eso no ocurre en un sistema descentralizado, pues sigue teniendo más unidades de procesamiento de información.
3. **Asignación de recursos equitativamente.** Un controlador centralizado necesita que se le asigne una gran parte de los recursos del sistema, que de otra forma (si no fuese centralizado) podrían ser asignados a otros agentes del sistema.

Al comienzo de la sección, y en capítulos anteriores, hemos puesto ejemplos de sistemas naturales que poseen elementos relativamente simples que procesan la información, que se comunican de forma local y que juntos pueden llevar a cabo una computación global de la información de forma paralela, algo que no podría ser llevado a cabo por cada uno de los individuos de forma individual, o mediante una combinación lineal de ellos.

A diferencia de las técnicas de computación en paralelo que existen actualmente (las cuales dividen el problema en diferentes tareas que son enviadas a los diferentes procesadores, y finalmente son llevados a un controlador centralizado donde la información se combina) en los sistemas que están realmente descentralizados ningún elemento que procesa la información (procesador) es más importante que otro y solamente tiene acceso a la información que esté en su entorno. Las señales se procesan y se propagan a través del sistema, respetando el alcance que posee la comunicación de sus elementos. Además, el almacenamiento de la información, su transmisión y su control están distribuidos a lo largo del sistema.

## 3.2. Objetivo

El objetivo de este trabajo es el de entender los mecanismos por los cuales la evolución puede descubrir métodos de computación emergente, analizar dichos mecanismos y comparar los resultados con los obtenidos por James P. Crutchfield y Melanie Mitchell en su obra "*The Evolution of Emergent Computation*" [2]. Aunque el estudio se realiza en un marco teórico, no deja de ser interesante, puesto que la esencia del fenómeno a estudiar se mantiene. Para ello necesitamos:

1. Una clase idealizada de sistema descentralizado en el cual el procesamiento de la información pueda surgir de las interacciones locales que realizan los individuos. Nosotros usamos los **autómatas celulares (ACs)**.
2. Una tarea que necesite el un procesamiento global de la información. En nuestro caso, se trata del **problema de la concentración**, el cual explicaremos a continuación.
3. Un modelo de evolución computacional idealizado. Para ello, hemos hecho uso de los **algoritmos genéticos**.

Para el estudio de la computación emergente utilizamos un AC unidimensional binario. Es uno de los sistemas más sencillos en los que se puede estudiar dicho fenómeno, ya que posee todas sus células dispuestas a lo largo de una línea, de manera que al ir evolucionando en el tiempo podemos tener una representación espacio-temporal de la evolución del autómatas, la cual podremos analizar posteriormente.

Hemos utilizado las mismas condiciones que utilizan Crutchfield y Mitchell en su artículo para poder realizar una comparación de los resultados. Y estas condiciones son:

- AC unidimensional binario, de 149 celdas con condiciones de contorno periódicas.
- Reglas totalísticas.
- Alcance de interacción  $r = 3$ .
- Una población de 100 reglas ( $\phi$ ) generadas de forma aleatoria.



La tarea que designamos es lo que hemos llamado antes **problema de la concentración**, y consiste en comprobar si el estado inicial del AC tiene más unos que ceros o viceversa, y evolucionar en consecuencia. Denominamos al umbral en la densidad de 1s como  $\rho_c = 1/2$ , y a  $\rho_0$  como la densidad de 1s en el estado inicial. Si  $\rho_0 > \rho_c$ , el comportamiento deseado es que el estado final de nuestro autómatas tenga todas las células “vivas” (todas las celdas del estado final serán 1s), mientras que si  $\rho_0 < \rho_c$  el estado final debería ser que todas las células estuviesen “muertas” (todas las celdas del estado final serán 0s). Además, debido al objetivo que queremos conseguir, necesitamos que el número de celdas en nuestro AC sea impar, de forma que nunca se pueda dar el caso  $\rho_0 = \rho_c$ . Por este motivo hemos escogido un número de celdas igual a 149, aunque podríamos haber escogido otro cualquiera nos hemos decantado por este por dos motivos. Primero, de esta forma mantenemos un paralelismo casi total con la simulación realizada por Mitchell y Crutchfield; y segundo, escogiendo esta cantidad de celdas podremos tener unos resultados claros sin tener un excesivo coste computacional.

### 3.3. Método

Como hemos dicho en el apartado anterior, hemos cogido las condiciones del experimento de Crutchfield y Mitchell. De forma que para el experimento disponemos de un AC unidimensional binario con 149 celdas, alcance  $r = 3$ , con reglas totalísticas, una población de 100 reglas generadas de forma aleatoria y cuya evolución viene dada por el método del algoritmo genético.

Primero generamos de forma aleatoria el conjunto de CIs.

En segundo lugar, si no se ha introducido un conjunto de reglas, se genera de forma aleatoria dicho conjunto, formado por un total de 100 reglas que serán las que hagan evolucionar nuestro AC.

A continuación hacemos evolucionar el autómatas para cada CI y para cada regla, un número de pasos de tiempo igual a  $\Delta t = 2N$  (siendo  $N$  el tamaño del autómatas), y almacenamos los estados de las celdas en cada paso de tiempo para luego poder calcular la función de ajuste  $F(\phi_r)$  de cada regla.

Acto seguido evaluamos el estado inicial y el final del AC (vemos cuántos 1s y cuántos 0s hay en cada estado y lo dividimos entre el número de celdas  $N$ ) de la siguiente forma:

$$\rho_0^{(i)} = \overline{S^{(i)}(t_0)} = \frac{1}{N} \sum_{j=0}^N s_j^{(i)}(t_0) \quad (\text{Densidad de 1s del estado inicial})$$

$$\rho_f^{(i)} = \overline{S^{(i)}(t_f)} = \frac{1}{N} \sum_{j=0}^N s_j^{(i)}(t_f) \quad (\text{Densidad de 1s del estado final})$$

Donde el superíndice  $(i)$  indica que se evalúa para la condición inicial  $i$ -ésima.

El número que dará  $\rho_0$  y  $\rho_f$  estará entre 0 y 1. Si  $\rho_0 > 0,5$  es que en el estado inicial hay más 1s que 0s, mientras que si  $\rho_0 < 0,5$  es que el estado inicial tiene más 0s que 1s. A la cantidad  $\rho_f$  la llamamos densidad de 1s en el estado final.

A los autómatas que consigan el objetivo se les asignará un valor de  $fit^{(i)}(\phi_r) = 1$ , mientras que a los que no lo consigan, se les asignará  $fit^{(i)}(\phi_r) = 0$ , donde  $fit^{(i)}(\phi_r)$  es un indicador de si la regla ha cumplido el objetivo en dicha condición inicial  $i$ -ésima (1) o no (0). Visto de forma más esquemática tenemos:

$$fit^{(i)}(\phi_r) = \begin{cases} 1 & \text{si } \rho_0 > 0,5 \text{ y } \rho_f = 1 \quad \text{ó} \quad \rho_0 < 0,5 \text{ y } \rho_f = 0 \\ 0 & \text{en otro caso} \end{cases}$$

Tras esto, la función de ajuste  $F(\phi_r)$  se calculará haciendo la suma de las  $fit^{(i)}(\phi_r)$  y promediándola sobre el número de CIs diferentes ( $n$ ).

$$F(\phi_r) = \frac{1}{n} \sum_{i=1}^n fit^{(i)}(\phi_r)$$

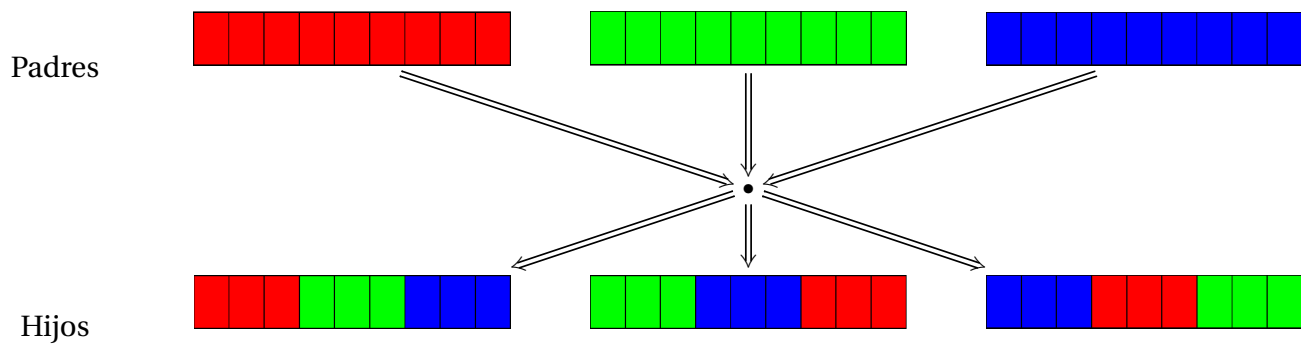
Repetimos el proceso para cada regla, guardando la calidad de cada regla en un vector de calidades  $F(\phi)$ , donde cada elemento corresponde a la calidad de una regla  $F(\phi_r)$ , donde el subíndice  $r$  corresponde a la regla  $r$ -ésima.

A continuación seleccionamos un cuarto del número total de reglas (25), siendo éstas las mejores de las 100. Estas 25 reglas son las que mantendremos para la siguiente generación, de esta forma nos aseguramos que el ajuste obtenido va a ser, como mínimo, alguno de los que obtuvimos con esas reglas. Estas reglas son las que se cruzarán para dar lugar a las nuevas generaciones.

Sabemos que la reproducción de las especies en la naturaleza se lleva a cabo mediante la reproducción asexual, que da lugar a un individuo idéntico al primero (salvo mutaciones), y la reproducción sexual, donde entran en juego dos individuos (macho y hembra) de la misma especie.

En la reproducción sexual cada individuo aporta su información genética, y ésta se recombina dando lugar a un nuevo individuo, quien comparte parte de los genes de sus progenitores. Estas son las formas mediante las cuales la naturaleza, hasta ahora, ha sido capaz de evolucionar haciendo que las especies mejor adaptadas sobrevivan. Pero... ¿qué ocurriría si la reproducción pudiese llevarse a cabo entre tres individuos diferentes? Esta pregunta es la que nos hemos hecho y hemos decidido comprobarlo mediante el algoritmo genético.

Como hemos comentado, el mecanismo de evolución que empleamos es el del algoritmo genético, pero en lugar de usar la convencional combinación genética de dos genes, hemos realizado cruces de tres genes, dividiendo cada gen en tres partes y combinándolas de la siguiente forma.



**Figura 3.1:** Esquema del cruce de las reglas.

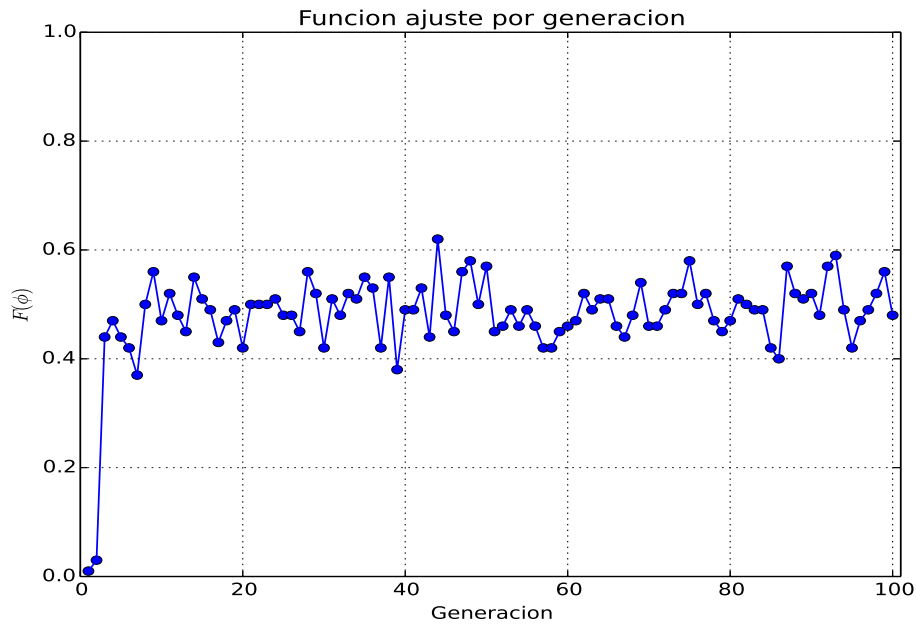
Tras obtener el nuevo conjunto de reglas las sometemos a una mutación con una probabilidad del 5 %. Esta mutación cambiará solamente el estado de un elemento de la regla (si es 0 lo muta a 1 y viceversa). De esta forma podemos evitar la caída en un mínimo local, el cual nos impediría obtener mediante la recombinación de reglas la mejor solución del espacio de las soluciones.

Una vez hecho esto, lo repetimos unas 100 generaciones aproximadamente obteniendo el ajuste de cada regla en cada generación y la representamos frente al número de generaciones, de forma que observamos cómo van mejorando las reglas en cada generación. Realizamos dos tandas de 100 generaciones.

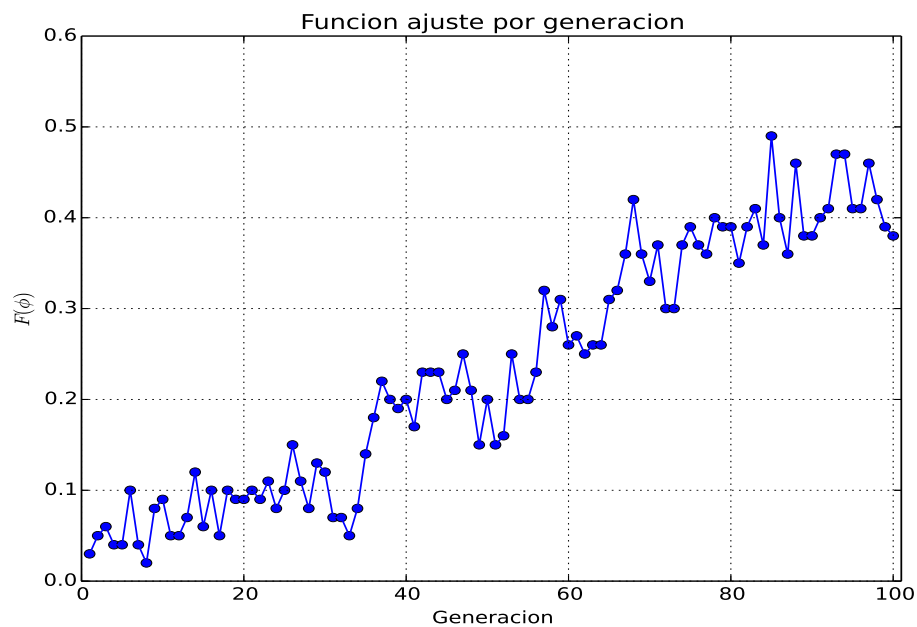
## 3.4. Resultados

### 3.4.1. Función de ajuste y “performance”

En los siguientes gráficos mostramos la evolución de la función de ajuste  $F(\phi)$  en cada generación.



(a) Tanda 1



(b) Tanda 2

**Figura 3.2:** Funciones de ajuste para las mejores reglas de cada generación frente al número de la generación.

Hemos representado la función de ajuste ( $F(\phi)$ ) de la mejor regla en cada generación para cada tanda de 100 generaciones, de forma que podemos observar la evolución de las reglas tras sufrir la recombinación y dar sus descendientes a lo largo de las generaciones.

Podemos observar las fluctuaciones de las funciones de ajuste en ambas tandas debido a que las reglas al estar evaluadas sobre 100 CIs dependen mucho de éstas.

En la primera tanda podemos observar como en la tercera generación la función de ajuste alcanza un valor próximo al 0,4, y a partir de ahí la  $F(\phi)$  fluctúa entre 0,6 y el 0,4, permaneciendo con un valor medio aproximadamente del 0,5, por lo que podría afirmarse que la calidad de estas reglas han permanecido aproximadamente constante.

En la segunda tanda de 100 generaciones podemos observar como, a pesar de existir dichas fluctuaciones por el mismo motivo anteriormente citado, existe un aumento progresivo en el valor de la función de ajuste, de forma que la calidad de las reglas va aumentando conforme pasan las generaciones.

Las dos gráficas representan dos casos típicos de evolución del AG. En el primero de ellos en las primeras generaciones se obtiene una regla bastante satisfactoria, pero tiene el inconveniente que sus “genes” dominan al resto y provocan que en conjunto la población evolucione poco más. El segundo caso, aún comenzando con individuos que son menos aptos pueden hacer que la población en su conjunto consiga una mejor solución.

Hemos realizado cuatro test de las mejores reglas de las generaciones que hemos considerado más representativas. Dicho test lo hemos realizado sobre 10 000 CIs, obteniendo las diferentes “performances” ( $P_{10^4}(\phi)$ ) que se muestran a continuación:

Reglas	$\phi_{44}^{(1)}$	$\phi_{100}^{(1)}$	$\phi_{85}^{(2)}$	$\phi_{100}^{(2)}$
$P_{10^4}(\phi)$	50,43 %	44,86 %	35,86 %	39,95 %

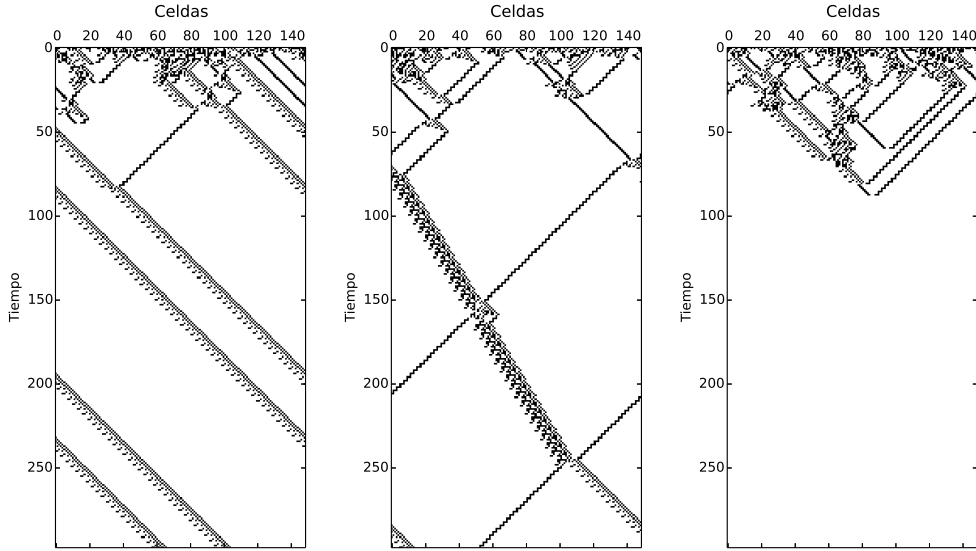
**Tabla 3.1:** “Performances” para dos reglas de cada tanda de 100 generaciones. Las dos mejores valoradas por la función de ajuste  $F(\phi)$  y las de la última generación para cada tanda. Los superíndices indican la tanda a la que pertenece, mientras que el subíndice indica la generación.

Observamos que para la primera tanda de generaciones, la mejor regla de la generación 44 (que es la que mejor función de ajuste tiene de esa tanda), nos da una “performance” del 50,43 %, mayor que la que nos da la última generación en esta misma tanda, 44,86 % aproximadamente. Como vemos, la calidad de la regla está próxima al 50 %, manteniéndose aproximadamente constante con el paso de las generaciones.

Mientras tanto, para la segunda tanda de generaciones podemos observar que la calidad de las reglas van aumentando conforme pasan las generaciones, como podemos ver en el gráfico (3.2 b) y en los valores obtenidos de las “performances” para la generación 85 y para la 100.

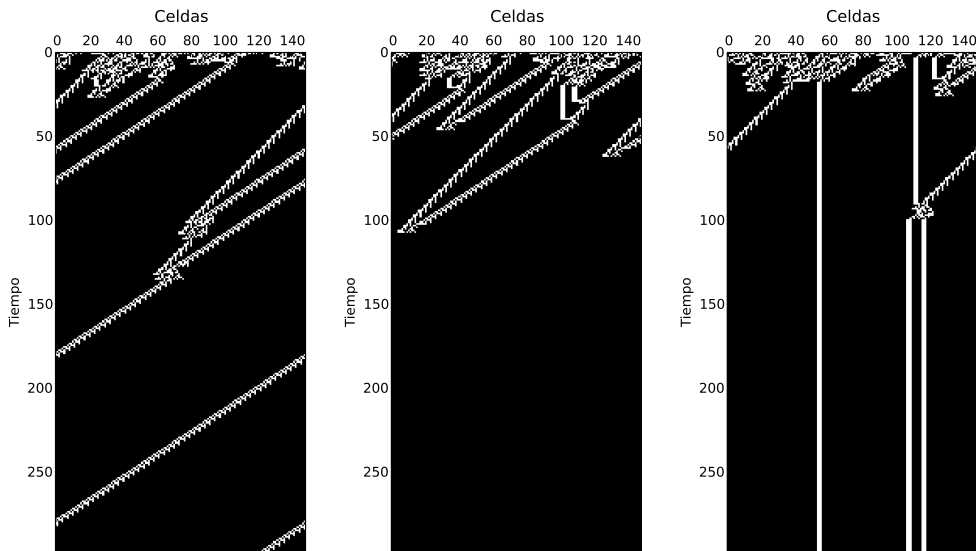
### 3.4.2. Diagramas de evolución espacio-tiempo

A continuación mostraremos algunas representaciones espacio-temporales de los ACs. Mostraremos indistintamente autómatas que hayan conseguido alcanzar el objetivo y autómatas que no, puesto que aunque no hayan alcanzado el objetivo, se pueden observar diferentes casos interesantes.



(a) Tanda 1:  $\rho_0 < 0,5$ ;  $\rho_f \neq 0$  (b) Tanda 1:  $\rho_0 < 0,5$ ;  $\rho_f \neq 0$  (c) Tanda 1:  $\rho_0 < 0,5$ ;  $\rho_f = 0$

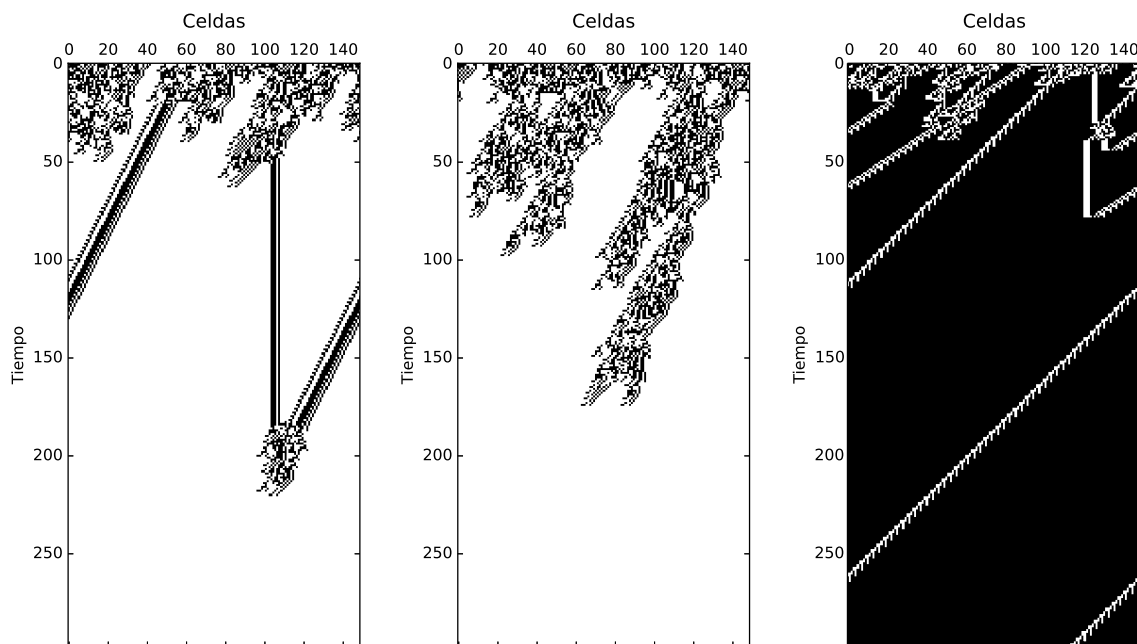
**Figura 3.3:** Evolución de la mejor regla de la **primera generación** para la primera tanda.



(a) Tanda 2:  $\rho_0 > 0,5$ ;  $\rho_f \neq 1$  (b) Tanda 2:  $\rho_0 > 0,5$ ;  $\rho_f = 1$  (c) Tanda 2:  $\rho_0 > 0,5$ ;  $\rho_f \neq 1$

**Figura 3.4:** Evolución de la mejor regla de la **primera generación** para la segunda tanda.

En las primeras generaciones para ambas tandas de reglas podemos observar diferentes tipos de interacciones. En estos diagramas podemos ver cómo el AC genera las partículas para comunicar las diferentes zonas del autómata, con la intención de alcanzar el objetivo.

(a) Tanda 1:  $\rho_0 < 0,5$ ;  $\rho_f = 0$ (b) Tanda 1:  $\rho_0 > 0,5$ ;  $\rho_f = 0$ (c) Tanda 2:  $\rho_0 > 0,5$ ;  $\rho_f \neq 1$ 

**Figura 3.5:** Evolución de la mejor regla de la **tercera generación** para la tanda 1 y la 2.

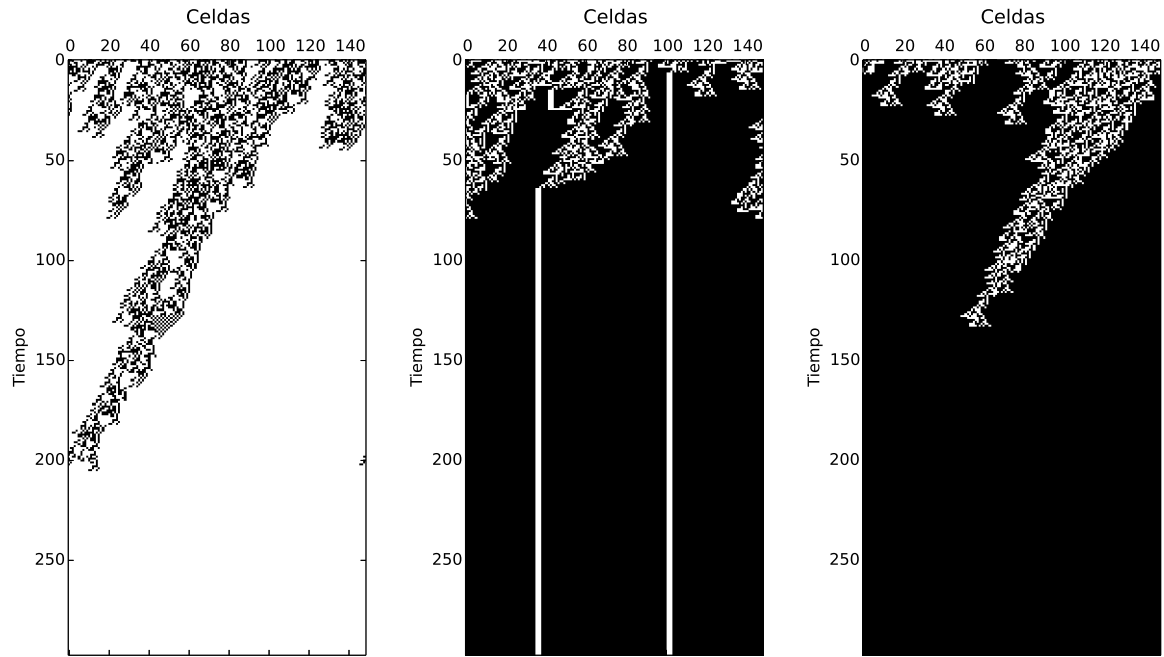
Podemos observar en la tercera generación para la primera tanda de reglas que los diagramas ya pertenecen a la clase IV, mientras que en el diagrama de la segunda tanda de reglas, el AC aún está enviando partículas para comunicarse con el resto del sistema.

En el primer diagrama observamos como tras la reproducción de una estructura, se generan un grupo que permanece estático en cada paso de tiempo y otro que avanza hacia la izquierda, llegando a interceptar a este grupo estático, aniquilándose mutuamente. En este caso la regla ha alcanzado el objetivo, puesto que la  $\rho_0 < 0,5$ .

En el segundo diagrama podemos observar como, tras evolucionar una estructura, todos los estados alcanzan el estado 0, pero la concentración de 1s en el estado inicial ( $\rho_0$ ) es mayor a 0,5, por lo que el autómata debería haber evolucionado a una situación en la que todas sus celdas fuesen 1, objetivo que como podemos comprobar, no ha conseguido.

El tercer diagrama es similar a los mostrados anteriormente.

Los siguientes diagramas son los correspondientes a la generación 44 y a la 85 de las tandas 1 y 2 respectivamente.

(a) Tanda 1:  $\rho_0 < 0,5 ; \rho_f = 0$ (b) Tanda 2:  $\rho_0 > 0,5 ; \rho_f \neq 1$ (c) Tanda 2:  $\rho_0 > 0,5 ; \rho_f = 1$ 

**Figura 3.6:** Evolución de la mejor regla de la **generación 44** para la tanda 1, y de la **generación 85** para la segunda tanda.

Podemos observar que en los tres diagramas el autómata lo que hace es repetir una estructura compleja, por lo que ambos tipos de diagramas pertenecen a la clase IV.

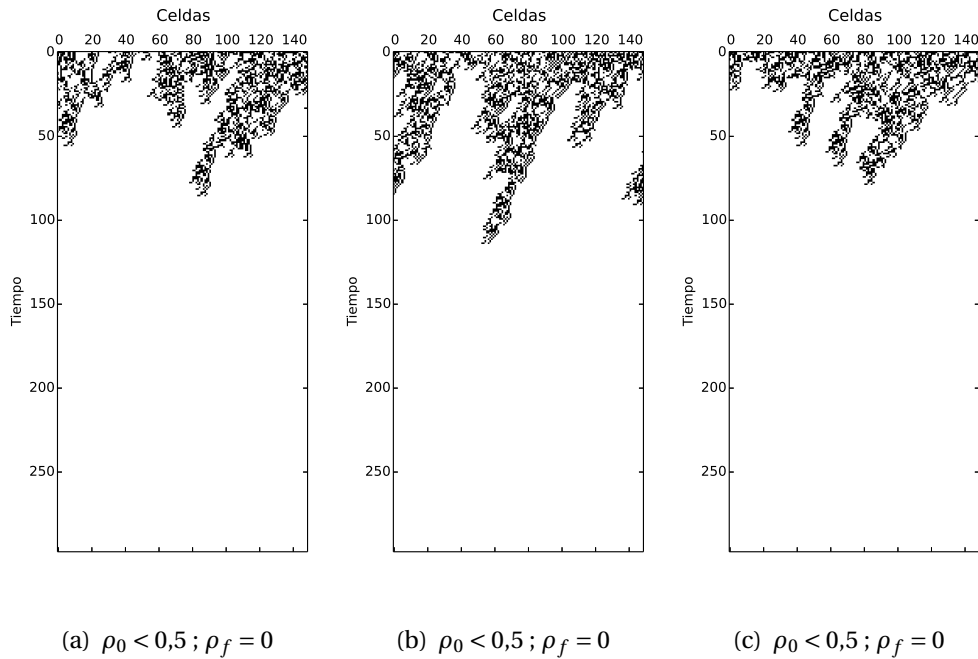
En el primer diagrama observamos como la estructura se va repitiendo hasta que tiende a desaparecer.

En el segundo, otro tipo de estructura se repite, hasta que en algunas regiones desaparecen, y en otras aparecen estados estacionarios.

En el tercer diagrama podemos observar cómo, análogamente al primer diagrama, se repite una estructura, reduciéndose hasta que desaparece. En estos tres diagramas solamente dos de ellos han alcanzado el objetivo que se les asignó.

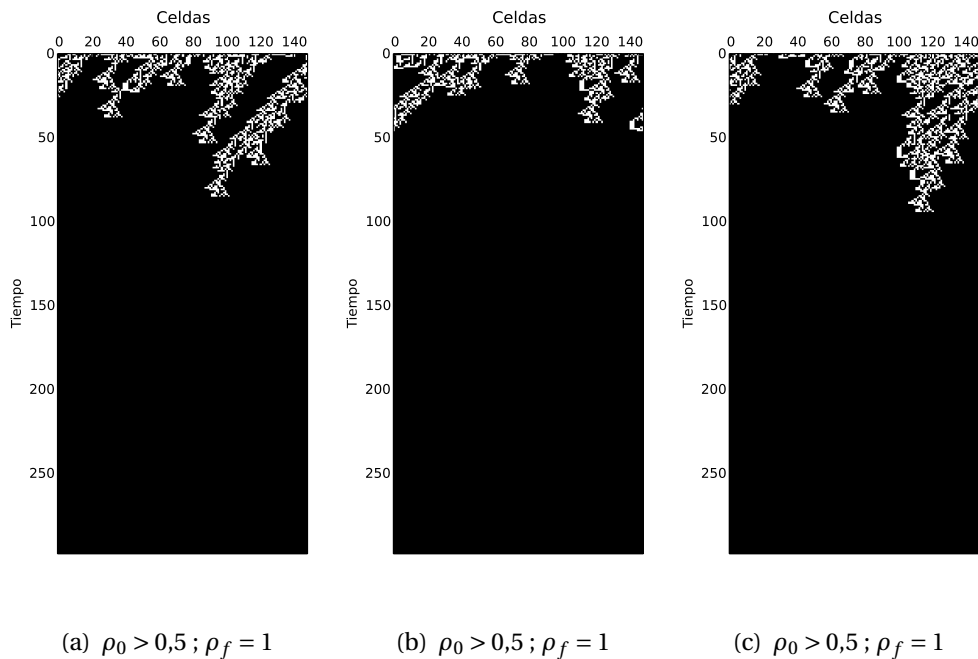


En las últimas generaciones de cada tanda observamos una evolución similar de los autómatas que en alguna de las generaciones anteriormente mostradas, donde observamos que se siguen repitiendo las estructuras complejas.



**Figura 3.7:** Evolución de la mejor regla de la **generación 100** para tres condiciones iniciales diferentes, de la primera tanda.

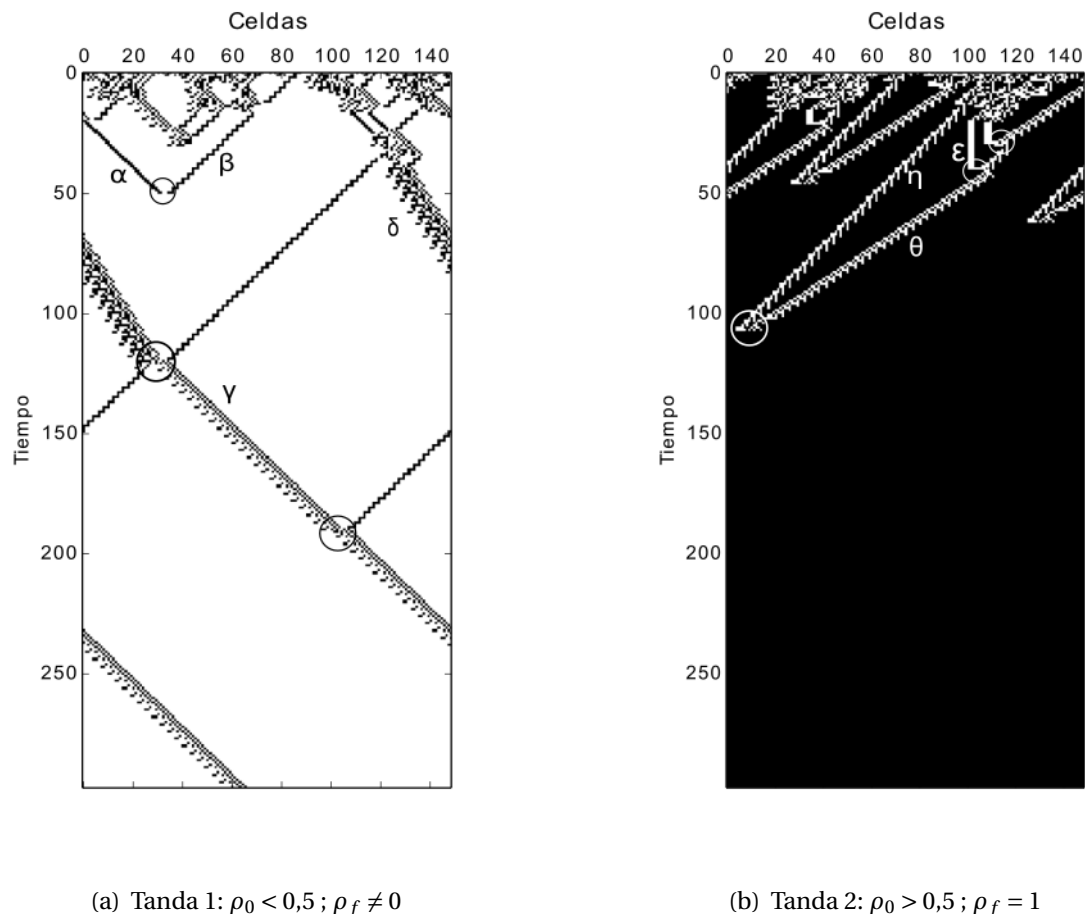
En estas últimas generaciones podemos observar como la comunicación que realiza el AC es mediante dichas estructuras complejas, que se transmiten y luego desaparecen.



**Figura 3.8:** Evolución de la mejor regla de la **generación 100** para tres condiciones iniciales diferentes, de la segunda tanda.

### 3.4.3. Análisis

Vamos a analizar los mecanismos que emplean los autómatas celulares en las primeras generaciones para comunicarse con zonas distantes del sistema.



**Figura 3.9:** Interacción entre partículas mensajeras. (a) Tanda 1, (b) Tanda 2.

En la figura de la izquierda (a) podemos observar como interaccionan diferentes partículas, dando lugar a otras nuevas, o aniquilándose entre ellas.

Hemos denominado a las partículas mensajeras de la figura (a) como “ $\alpha$ ”, “ $\beta$ ”, “ $\gamma$ ” y “ $\delta$ ”.

Podemos observar los tres círculos marcados en la figura, los cuales representan claramente tres interacciones diferentes entre las partículas.








El círculo superior izquierda muestra la interacción entre una partícula  $\alpha$  y una  $\beta$ , aniquilándose mutuamente. El siguiente círculo muestra la interacción entre una partícula  $\beta$  y una  $\delta$ , dando lugar a una partícula  $\gamma$  y a una  $\beta$ . El último círculo muestra la interacción entre una partícula  $\gamma$  y una  $\beta$ , siendo esta última aniquilada al interactuar con la partícula  $\gamma$ .

En la segunda imagen (b) podemos observar como el autómata comienza evolucionando con lo que parece ser la estructura compleja que presentará en las últimas generaciones, para luego decaer en tres tipos de partículas mensajeras, a las cuales hemos llamado “ $\epsilon$ ”, “ $\theta$ ” y “ $\eta$ ”.

En los tres círculos que hemos marcado en el diagrama podemos observar tres interacciones diferentes entre las partículas.

En el círculo superior derecho podemos observar la interacción entre la partícula  $\epsilon$  (partícula estacionaria) y la partícula  $\theta$ , dando lugar a la partícula  $\eta$ . En el siguiente círculo observamos la interacción entre la partícula  $\epsilon$  y la partícula  $\eta$ , generando una partícula  $\theta$ . La última interacción observamos la interacción entre las partículas  $\theta$  y  $\eta$ , aniquilándose mutuamente.

En la siguiente tabla mostramos las diferentes interacciones para los dos diagramas.

	Tanda 1				Tanda 2		
Partículas							
	$\alpha$	$\beta$	$\gamma$	$\delta$	$\epsilon$	$\eta$	$\theta$
Interacciones	$\gamma + \beta \rightarrow \gamma$				$\epsilon + \theta \rightarrow \eta$		
	$\beta + \delta \rightarrow \gamma$				$\epsilon + \eta \rightarrow \theta$		
	$\alpha + \beta \rightarrow \emptyset$				$\eta + \theta \rightarrow \emptyset$		

**Tabla 3.2:** Tabla sobre el tipo de partículas mensajeras y sus interacciones. El símbolo  $\emptyset$  indica que las partículas se aniquilan al interactuar.

Una regla idónea transformará cualquier configuración inicial a todas las celdas 1 (ó 0) y para ello la “química de partículas” tendría que tener reglas de aniquilación para todas ellas. La imposibilidad para resolver el problema por parte de las reglas de la Tanda-1 se resume en la existencia de la partícula  $\gamma$  que no tiene ninguna partícula con la que aniquilarse. La Tanda-2, en este sentido es más sofisticada ya que las partículas detectadas aquí si que pueden llegar a aniquilarse entre ellas.

Llegado a este punto vamos a comparar las reglas obtenidas por nuestro AG con las obtenidas por Mitchell y Crutchfield, y con la regla denominada regla GKL que es la que mejor resuelve el problema de la concentración [2]. Esta comparación la realizamos con las reglas aplicadas al caso en el que el tamaño del autómata es de 149 celdas.

CA ( $r = 3$ )	Regla en hexadecimal	Símbolo	$P_{10^4}(\phi)$
White	28834f913e4f146211592eed05dcd43a	$\phi_W$	0.504
Black	d575bc35c51f130ba66ade17c09fb89b	$\phi_B$	0.400
Particle (M&C)	031001001fa00013331f9fff5975ffff	$\phi_{17083}$	0.755
Particle (M&C)	0504058705000f77037755837bffb77f	$\phi_{100}$	0.769
GKL	005f005f005f005f005fff5f005fff5f	$\phi_{GKL}$	0.816

**Tabla 3.3:** Comparación de “performances” de las reglas obtenidas por nuestro algoritmo genético ( $\phi_W$  y  $\phi_B$ ), con las obtenidas por Mitchell y Crutchfield (M&C) y con la GKL.

Para convertir el código hexadecimal en binario, y poder así utilizar la regla para realizar computaciones, primero hay que convertirlas a decimal, y luego pasar dicho número decimal a binario.

Para pasar de un número hexadecimal a decimal simplemente hay que hacer la suma sobre el producto del número decimal que le corresponde a ese carácter hexadecimal por 16 elevado a un número, y dicho número es la posición que ocupa dicho carácter hexadecimal (contando de derecha a izquierda, y siendo la primera posición 0). Aquí tenemos un ejemplo:

$$7DE = (7 \cdot 16^2) + (13 \cdot 16^1) + (14 \cdot 16^0) = 2014$$

Para convertir un número decimal a binario, lo único que hay que hacer es coger ese número e ir dividiéndolo entre dos hasta que ya el numerador no sea divisible entre 2. Luego, el orden del número es el inverso al de la realización de las divisiones, es decir, el primer número será el último cociente que obtuvimos, el siguiente será el resto de la división anterior, y así sucesivamente, donde el último dígito será el primer resto obtenido. Aquí ponemos un ejemplo:

20 : 2	0	Luego: $20_{10} \rightarrow 10100_2$
10 : 2	0	
5 : 2	1	
2 : 2	0	
1		

### 3.5. Conclusiones

En este trabajo nos hemos puesto en contacto con los sistemas complejos y con los métodos de los autómatas celulares y los algoritmos genéticos, además de tratar el problema de la concentración y de la computación emergente. Además, hemos realizado una profunda investigación al respecto en artículos de diferentes autores (artículos como el de Mitchell y Crutchfield [2], de Wolfram [9] y [10], Hiroki Sayama [8], etc). Podemos resumir el trabajo realizado en los siguientes puntos:

1. Los códigos han sido creados totalmente desde cero, de forma original, escritos en lenguaje *Python* y ejecutados en el *cluster Azahar* de la Universidad de Sevilla. Los tiempos de cálculo requeridos para esta tarea han sido aproximadamente de 20 minutos por simulación para el código principal (A.1) y para el test (A.2), habiéndose llevado acabo 200 simulaciones (100 por cada tanda de 100 generaciones, habiendo realizado dos tandas únicamente), además del tiempo empleado en escribir los algoritmos. Las especificaciones del cluster aparecen en el siguiente enlace:

<https://azaharcluster.wordpress.com/2013/01/29/descripcion-del-cluster-azahar/>

Aunque los tiempos de cálculo pueden mejorarse si el código se implementa en lenguaje C/C++ y/o si se emplea programación en paralelo, puesto que el código es altamente paralelizable, hemos elegido realizarlo en *Python* debido a sus ventajas, tales como: sintaxis más simple, “tipado” dinámico y gran popularidad en el mundo de la programación (es el sexto lenguaje más utilizado), entre otros.

2. Hemos estudiado el problema de la concentración mediante el uso de los ACs y los AGs, siendo capaces de encontrar reglas que cumplen los objetivos ( $\phi_W$  clasifica adecuadamente los casos donde  $\rho_0 < 0,5$ , mientras que  $\phi_B$  clasifica de forma correcta los casos donde  $\rho_0 > 0,5$ ).

Además hemos analizado el mecanismo por el cual el AC se comunica con las zonas más distantes de su sistema, siendo este mecanismo la emisión de partículas, las cuales al interactuar con otras generan otras partículas o se aniquilan.

3. En este problema hemos introducido una idea original, que es la de emplear un algoritmo genético con tres progenitores, además de haber ido cambiando las CIs en cada generación, cuya ventaja es la de que la calidad de las reglas obtenidas (de media) no depende de las CIs. Aunque hemos realizado únicamente dos experiencias (de 100 generaciones cada una), hemos conseguido unas “performances” entorno al 40-50 %, lo que aporta luz a este método para seguir trabajando sobre él, puesto que con más experimentos podremos obtener una mayor calidad en las reglas.
4. Hemos realizado la comparación entre las reglas obtenidas por nuestro algoritmo genético con las obtenidas por Mitchell y Crutchfield, destacando que aunque la calidad de nuestras reglas es algo menor, nuestro procedimiento también es adecuado para la obtención de las mejores reglas, puesto que, volviendo a destacarlo, hemos podido realizar únicamente dos experimentos y aún así hemos obtenido una calidad entorno al 40-50 %.
5. Para finalizar marcamos unas líneas futuras por las cuales podemos seguir investigando y trabajando: aumentar el número de experimentos realizados que mejorará claramente los resultados obtenidos, estudiar la propagación de la información mediante las partículas de la clase IV o mantener las CIs sin alterarlas en cada generación.

# Apéndice A

## Códigos

A continuación encontramos los diferentes códigos que hemos desarrollado para realizar el trabajo.

También los podemos encontrar en mi github:

<https://github.com/DanielLopezCoto/emergent-computation>

### A.1. Computación Emergente

El código que mostramos a continuación es el que hemos desarrollado para obtener las diferentes reglas y hacerlas evolucionar en cada nueva generación.

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  """
4  Autor: Daniel Lopez Coto
5
6  Este programa pretende reproducir los resultados de la computacion emergente usando
7  los
8  metodos de los automatas celulares unidimensionales.
9
10 Informacion util:
11     - r = 3 (vecindad de orden 3)
12     - D = 1 (automatas unidimensionales)
13     - k = 2 (dos posibles estados: 0 o 1)
14     - 2**(2*r+1) = Numero de bits necesarios para las reglas (en este caso 128 bites)
15     - Numero de reglas: 100
16     - Uso de algoritmos geneticos
17     - Reglas de transicion totalisticas
18     - Tamano de los automatas (size) = 149
19 """
20
21 import numpy as np
22 import random as rd
23 import csv
24 from matplotlib import pylab
25 import sys
26
27 import time as TIEMPO
```

```

29 """ Definimos un decorador, que nos dara el tiempo de ejecucion de la funcion
decorada """
31
32 def cronometro(funcion):
33     def fun_a_ejecutar(*args):
34         t1 = TIEMPO.time()
35         resultado = funcion(*args)
36         t2 = TIEMPO.time()
37         T = t2 - t1
38         nombre = funcion.__name__
39         print("Tiempo de '%s': %g segundos" %(nombre,T))
40         return resultado
41     return fun_a_ejecutar
42
43 """ Clase automata que es la que creara las reglas, o en caso de haberas introducido
nosotros, las maneja; calculara el ajuste (fitness) de dichas reglas y las
45 ira modificando """
46
47 class Automata:
48
49     def __init__(self, numeroReglas=100, initSize=149, tiempo=None, initVector=None,
initReglas=None):
50
51         """ Inicializamos los valores que vamos a necesitar en el programa.
Los parametros de entrada que permite son: Numero; tamaño del automata (numero
53 de celdas); set inicial de condiciones iniciales y set de reglas, respectivamente.
'setReglas' es una lista que contiene las reglas (lista de listas).
55 'setVectores' es una lista que contiene las configuraciones iniciales de las
celdas (lista
de listas)"""
56
57         self.size = initSize          # Numero de celdas que tendra el automata.
58         self.noReglas = numeroReglas  # Numero de reglas.
59         self.time = tiempo
60         if self.time == None:
61             self.time = initSize * 2
62
63         if initReglas == None:         # Si no hay reglas inicialmente, las creamos.
64             self.setReglas = tuple(self.crearReglas(self.noReglas))
65         else:                          # Si ya hemos introducido las reglas, las inicializamos.
66             self.setReglas = tuple(initReglas)
67
68         if initVector == None:         # Si no hay un vector con los valores iniciales,
69             # lo creamos.
70             self.setVectores = condIniciales(self.size)
71         else:                          # Si lo hay, lo inicializamos.
72             self.setVectores = initVector
73
74
75     def crearReglas(self, noReglas):
76         Rules = list(np.zeros(noReglas)) # Lista que almacena las diferentes reglas.
77
78         for i in xrange(noReglas):
79             rule = []                  # Lista que contendra a la regla i-esima.
80             for j in xrange(128):
81                 a = rd.random()       # En cada iteracion generamos un numero aleatorio entre 0 y
1

```

```

83         if a > 0.5:
84             rule.append(1)
85         else:
86             rule.append(0)
87
88     Rules[i] = rule
89
90     return Rules      # Devuelve una lista de reglas, donde cada elemento es
91                       # una regla de 128 elementos 0s y 1s (es decir, una lista
92                       # de 128 elementos).
93
94 def cambiarEstado(self, regla, tiempo):
95     """ Este metodo recibe como entrada un entero que corresponde a la cantidad de
96     pasos de tiempo que se van a evaluar los estados de las celulas para cada regla."""
97
98     self.setEstados = self.setVectores[:] # Lista que contiene el conjunto de
99     estados para la regla
100     # introducida como parámetro,
101     # para cada CI, y para cada paso de tiempo.
102     numEsta2Iniciales = len(self.setEstados) # Almacena el numero de estados
103     iniciales
104     # que correspondera con el numero de CIs.
105
106     M_r = [[None]*tiempo for s in xrange(numEsta2Iniciales)] # Matriz que almacena
107     los estados para cada
108     # paso de tiempo, y para cada CI. Las filas son las
109     diferentes
110     # CIs y las columnas los diferentes estados de tiempo.
111     # M_r[i] todos los estados de tiempo de la CI i-esima.
112
113     for i in xrange(numEsta2Iniciales): # Iteramos sobre las Configuraciones
114     Iniciales.
115         estadoActual = self.setEstados[i][:] # Escogemos la CI i-esima.
116         nuevoEstado = estadoActual[:]
117         M_r[i][0] = estadoActual[:]
118
119         for t in xrange(1,tiempo): # Iteramos sobre cada paso de tiempo.
120             for j in xrange(-3,self.size-3): # Iteramos sobre las posiciones.
121                 """ Codificamos la configuracion de vecinos en binario, y lo convertimos en
122                 un
123                 numero. Dicho numero sera la posicion de la regla que estamos evaluando,
124                 y el valor que este almacenado en dicha posicion, sera asignado al nuevo
125                 estado de la celula j."""
126                 binario = '0b'+str(estadoActual[j-3])+str(estadoActual[j-2])+\\
127                     str(estadoActual[j-1])+str(estadoActual[j])+\\
128                     str(estadoActual[j+1])+str(estadoActual[j+2])+str(estadoActual[j+3])
129                 nuevoEstado[j] = regla[int(binario,2)]
130                 estadoActual = nuevoEstado[:]
131                 M_r[i][t] = [float(s) for s in estadoActual]
132
133     return M_r      # Devuele la matriz con todas las configuraciones de cada paso
134     de tiempo, para
135     # cada CI.
136
137 def fitnessFun(self, confVectores):

```



```

133     """ Este metodo recibe como entrada el set de configuraciones de las celulas
134     en cada instante de tiempo para una regla concreta."""
135
136     fit = []          # Lista que almacena los ajustes para las diferentes CIs
137     numElementos = self.size # Tamaño del automata
138     numCIs = len(confVectores) # Numero de CIs
139     pasosTiempo = len(confVectores[0]) # Numero de pasos de tiempo
140     for i in xrange(numCIs):
141         inicial = confVectores[i][0]
142         sumaini = float(sum(inicial))/numElementos # Nos dice si la configuracion
143         inicial tiene mas ceros o unos
144         final = confVectores[i][-1] # Obtenemos el ultimo paso de tiempo para la CI i-
145         esima
146         sumafin = float(sum(final))/numElementos # Fraccion de unos en el estado final
147         if sumaini < 0.5 and sumafin == 0 or sumaini > 0.5 and sumafin == 1: #
148         Si el estado inicial tiene mas 0 que unos, cambia fit por 1-fit
149             fit.append(1)
150         else:
151             fit.append(0)
152
153     ajuste = float(sum(fit))/len(fit) # Hace una media del valor del ajuste para las
154                                         # diferentes CIs
155     return ajuste #,id_mejor
156
157 @cronometro
158 def algoritmoGenetico(self, tiempo):
159     """ Llama al metodo 'cambiarEstado' para crear los diferentes estados para cada
160     intervalo
161     de tiempo en cada CI. Llama a la 'fitnessFun' para calcular el grado de ajuste de
162     la regla
163     que queramos ver en ese momento. Iteramos sobre cada regla para obtener un vector
164     con la calidad de cada una, y posteriormente poder realizar la seleccion de las
165     mejores
166     y eliminar las demas, de forma que podamos realizar el 'crossover' de las
167     distintas reglas,
168     asegurandonos obtener en cada nueva generacion reglas mejores."""
169
170     contador = [] # Lista que almacena el numero de la regla que cumple la
171                     # seleccionada
172     limRegla = self.noReglas/4
173     newReglas = []
174     self.bondad = []
175     self.M_R = [[None]*tiempo for s in xrange(len(self.setVectores))] \
176                 for k in xrange(self.noReglas)]
177     # Lista que contendra todos los estados para cada paso de tiempo para cada CI de
178     # cada regla.
179     # El elemento M_R[r] es el conjunto de todos los pasos de tiempo para todas las
180     # CIs en la regla r-esima.
181     #self.id_mejor_CI = [] # lista que contendrá índice de la CI que mejor ajusta la
182     # regla r
183
184     for r in xrange(self.noReglas): # Evaluamos la bondad de cada regla.
185         self.M_R[r] = self.cambiarEstado(self.setReglas[r], self.time)
186         salidaFitnessFun = self.fitnessFun(self.M_R[r])
187         self.bondad.append(salidaFitnessFun) #[0]
188         #self.bondad.append(self.fitnessFun(self.M_R[r]))
189         #self.id_mejor_CI.append(salidaFitnessFun[1])

```

```

181 maxima = max(self.bondad) # Mira cual es el maximo de la bondad de las reglas
183 contador.append(self.bondad.index(maxima))
185 print contador
187 for r in xrange(self.noReglas):
189     if self.bondad[r] < maxima and self.bondad[r] >= 0.8*maxima:
191         contador.append(r)
193         if len(contador) == limRegla:
195             break
197 while len(contador) < limRegla:
199     for r in xrange(self.noReglas):
201         if self.bondad[r] > 0.5*maxima and self.bondad[r] < 0.8*maxima:
203             contador.append(r)
205             else:
207                 continue
209                 if len(contador) == limRegla:
211                     break
213                 if len(contador) != limRegla:
215                     for r in xrange(self.noReglas):
217                         a = int(rd.random()*99)
219                         if a in contador:
221                             continue
223                         else:
225                             contador.append(a)
227                             if len(contador) == limRegla:
229                                 break
231 print contador #borrar luego
233 for i in contador:
235     newReglas.append(self.setReglas[i]) # Añade las reglas

# Escribimos las reglas actuales en un fichero
ruls = open('reglas.txt', 'w')
out = csv.writer(ruls, delimiter = '\t')
out.writerows(self.setReglas)
ruls.close()

"""Ahora hay que hacer el crossover de reglas"""
for r in xrange(-1,limRegla-1):
    a = newReglas[r][:]
    b = newReglas[r+1][:]
    c = newReglas[r+2][:]

    aNew = a[:]
    bNew = b[:]
    cNew = c[:]

    aNew[42:84],aNew[84:] = c[42:84], b[84:]
    bNew[42:84],bNew[84:] = a[42:84], c[84:]
    cNew[42:84],cNew[84:] = b[42:84], a[84:]

# Añade las nuevas reglas
newReglas.append(aNew)
newReglas.append(bNew)
newReglas.append(cNew)

# Probabilidad de mutacion
for r in xrange(len(newReglas)):

```

```

239     if rd.random() <= 0.05:
240         pos_mut = int(rd.random()*127)
241         if newReglas[r][pos_mut] ==1:
242             newReglas[r][pos_mut] = 0
243         else :
244             newReglas[r][pos_mut] = 1
245
246     # Escribimos las nuevas reglas
247     n_ruls = open('new_reglas.txt','w')
248     n_out = csv.writer(n_ruls,delimiter = '\t')
249     n_out.writerow(newReglas)
250     n_ruls.close()
251
252 def condIniciales(size=149,no_CIs=100):
253     """ Creamos un conjunto de 100 configuraciones iniciales para cada celula.
254     Los estados posibles son 0 o 1, repartidos de forma aleatoria siguiendo una
255     distribucion uniforme en el intervalo (0,1)"""
256
257     setCIs = [] # Lista vacia que contendra el conjunto
258                 # de las 100 configuraciones iniciales.
259     for i in xrange(no_CIs):
260         vectIni = [] # Lista que contendra la configuracion inicial i-esima.
261         for j in xrange(size):
262             if rd.random() > 0.5:
263                 vectIni.append(1)
264             else:
265                 vectIni.append(0)
266
267         setCIs.append(vectIni)
268     C = open('setCI.txt','w')
269     Cout = csv.writer(C,delimiter = '\t')
270     Cout.writerow(setCIs)
271     C.close()
272     return setCIs
273
274 class AutomImagen:
275
276     def __init__(self,Objeto=None): # Objeto sera la clase automata con sus parametros
277                                     # iniciados.
278
279         if Objeto == None:
280             sys.exit()
281
282         #self.Objeto = Objeto
283         Objeto = Objeto
284         self.nCI = len(Objeto.setVectores)
285         time = Objeto.time
286         Objeto.algoritmoGenetico(time)
287         self.bondad = Objeto.bondad
288         self.m_r = Objeto.M_R
289         #self.id_CI = Objeto.id_mejor_CI
290         self.dibujAutom()
291
292     @cronometro
293     def dibujAutom(self):
294         maximo = max(self.bondad)
295         print maximo
296         for r in xrange(len(self.m_r)):

```

```

295         if self.bondad[r] == maximo:
296             regla = r
297             #ID = self.id_CI[r]
298             break
299     for i in xrange(self.nCI):
300         c = (regla, i) # tupla que dara el numero de la regla y de la CI para guardar
301         la imagen.
302         matrizEstados = self.m_r[r][i]
303         pylab.matshow(matrizEstados, cmap='binary')
304         pylab.title("Celdas")
305         pylab.ylabel("Tiempo")
306         pylab.savefig("Regla%i_CI%i.eps" %c)
307         pylab.close()
308         # Aquí salvamos el automata seleccionado
309         f = open('automata_R%i_CI%i.txt' %c, 'w')
310         fout = csv.writer(f, delimiter = '\t')
311         fout.writerow(matrizEstados)
312         f.close()
313
314     """Esta parte del codigo sirve para que el programa se ejecute solo si el modulo
315     es llamado como programa y no al importarlo."""
316
317     if __name__ == '__main__':
318
319         print "¿Tienes reglas que importar? ([y]/n):"
320         respuesta = raw_input()
321
322         if respuesta != 'n':
323             #print "Escribe el nombre del archivo con la extension."
324             #archivo = raw_input()
325             archivo = 'new_reglas.txt'
326             reglasImportadas = open(archivo, 'r')
327             setreglas = reglasImportadas.readlines()
328             nreglas = len(setreglas)
329             listaReglas = []
330             for l in xrange(nreglas):
331                 regla = setreglas[l].split()
332                 for i in xrange(len(regla)):
333                     regla[i]=int(regla[i])
334                 listaReglas.append(regla)
335             automata = Automata(initReglas = listaReglas)
336
337         else:
338             automata = Automata()
339
340     AutomImagen(automata)

```

## A.2. Test

El código para hacer el test a la mejor regla de todas es similar al de la sección anterior, pero modificado para leer una sola regla y evaluarla sobre diez mil condiciones iniciales.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  """
4  Autor: Daniel Lopez Coto
5
6  Este programa testea la regla seleccionada sobre 10000 CIs
7  """
8
9  import numpy as np
10 import random as rd
11 import csv
12 from matplotlib import pylab
13 import sys
14
15 import time as TIEMPO
16
17 """ Definimos un decorador, que nos dara el tiempo de ejecucion de la funcion
18 decorada """
19
20 def cronometro(funcion):
21     def fun_a_ejecutar(*args):
22         t1 = TIEMPO.time()
23         resultado = funcion(*args)
24         t2 = TIEMPO.time()
25         T = t2 - t1
26         nombre = funcion.__name__
27         print("Tiempo de '%s': %g segundos" %(nombre,T))
28         return resultado
29     return fun_a_ejecutar
30
31 @cronometro
32 def condIniciales(size=149,no_CIs=10000):
33     """ Creamos un conjunto de 100 configuraciones iniciales para cada celula.
34     Los estados posibles son 0 o 1, repartidos de forma aleatoria siguiendo una
35     distribucion uniforme en el intervalo (0,1)"""
36
37     setCIs = [] # Lista vacia que contendra el conjunto
38                # de las 100 configuraciones iniciales.
39     for i in xrange(no_CIs):
40         vectIni = [] # Lista que contendra la configuracion inicial i-esima.
41         for j in xrange(size):
42             if rd.random() > 0.5:
43                 vectIni.append(1)
44             else:
45                 vectIni.append(0)
46
47         setCIs.append(vectIni)
48
49     return setCIs
50
51 class Automata:

```

```

54 def __init__(self, numeroReglas=1, initSize=149, tiempo=None, initVector=None, initReglas
55     =None):
56
57     """ Inicializamos los valores que vamos a necesitar en el programa.
58     Los parametros de entrada que permite son: Numero; tamaño del automata (numero
59     de celdas); set inicial de condiciones iniciales y set de reglas, respectivamente.
60     'setReglas' es una lista que contiene las reglas (lista de listas).
61     'setVectores' es una lista que contiene las configuraciones iniciales de las
62     celdas (lista
63     de listas)"""
64
65     self.size = initSize      # Numero de celdas que tendra el automata.
66     self.noReglas = numeroReglas # Numero de reglas.
67     self.time = tiempo
68     if self.time == None:
69         self.time = initSize * 2
70
71     if initReglas == None:      # Si no hay reglas inicialmente, salimos
72         print "No hay reglas que testear."
73         sys.exit()
74     else:                      # Si ya hemos introducido las reglas, las inicializamos.
75         self.setReglas = tuple(initReglas)
76
77     if initVector == None:      # Si no hay un vector con los valores iniciales,
78         # lo creamos.
79         self.setVectores = condIniciales(self.size)
80     else:                      # Si lo hay, lo inicializamos.
81         self.setVectores = initVector
82
83     self.M_R = self.cambiarEstado(self.setReglas, self.time)
84     salidaFitnessFun = self.fitnessFun(self.M_R)
85     self.bondad = salidaFitnessFun#[0]
86     #self.id_mejor_CI = salidaFitnessFun[1]
87
88 @cronometro
89 def cambiarEstado(self, regla, tiempo):
90     """ Este metodo recibe como entrada un entero que corresponde a la cantidad de
91     pasos de tiempo que se van a evaluar los estados de las celulas para cada regla."""
92
93     self.setEstados = self.setVectores # Lista que contiene el conjunto de estados
94     # para la regla
95     # introducida como parámetro,
96     # para cada CI, y para cada paso de tiempo.
97     numEsta2Iniciales = len(self.setEstados) # Almacena el numero de estados
98     # iniciales
99     # que correspondera con el numero de CIs.
100
101     M_r = [[None]*tiempo for s in xrange(numEsta2Iniciales)] # Matriz que almacena
102     # los estados para cada
103     # paso de tiempo, y para cada CI. Las filas son las
104     # CIs y las columnas los diferentes estados de tiempo.
105     # M_r[i] todos los estados de tiempo de la CI i-esima.
106
107     for i in xrange(numEsta2Iniciales): # Iteramos sobre las Configuraciones

```

```

106 Iniciales.
    estadoActual = self.setEstados[i][:] # Escogemos la CI i-esima.
    nuevoEstado = estadoActual[:]
    M_r[i][0] = estadoActual[:]

108
    for t in xrange(1,tiempo): # Iteramos sobre cada paso de tiempo.
110         for j in xrange(-3,self.size-3): # Iteramos sobre las posiciones.
            """ Codificamos la configuracion de vecinos en binario, y lo convertimos en
un
112         numero. Dicho numero sera la posicion de la regla que estamos evaluando,
            y el valor que este almacenado en dicha posicion, sera asignado al nuevo
114         estado de la celula j."""
            binario = '0b'+str(estadoActual[j-3])+str(estadoActual[j-2])+\\
116                     str(estadoActual[j-1])+str(estadoActual[j])+\\
                     str(estadoActual[j+1])+str(estadoActual[j+2])+str(estadoActual[j+3])
118             nuevoEstado[j] = regla[int(binario,2)]
            estadoActual = nuevoEstado[:]
120             M_r[i][t] = [float(s) for s in estadoActual]

122         return M_r # Devuele la matriz con todas las configuraciones de cada paso
de tiempo, para
            # cada CI.

124
126 @cronometro
def fitnessFun(self,confVectores):
128     """ Este metodo recibe como entrada el set de configuraciones de las celulas
en cada instante de tiempo para una regla concreta."""

130
    fit = [] # Lista que almacena los ajustes para las diferentes CIs
132     numElementos = self.size # Tamaño del automata
    numCIs = len(confVectores) # Numero de CIs
134     pasosTiempo = len(confVectores[0]) # Numero de pasos de tiempo
    for i in xrange(numCIs):
136         inicial = confVectores[i][0]
        sumaini = float(sum(inicial))/numElementos # Nos dice si la configuracion
138         inicial tiene mas ceros o unos
        final = confVectores[i][-1] # Obtenemos el ultimo paso de tiempo para la CI i-
esima
        sumafin = float(sum(final))/numElementos # Fraccion de unos en el estado final
140         if sumaini < 0.5 and sumafin == 0 or sumaini > 0.5 and sumafin == 1:
            fit.append(1)
142         else:
            fit.append(0)

144
    ajuste = float(sum(fit))/len(fit) # Hace una media del valor del ajuste para las
146                                     # diferentes CIs
    return ajuste #,id_mejor

148
150 class AutomImagen:
152
    def __init__(self,Objeto=None): # Objeto sera la clase automata con sus parametros
iniciados.

154
    if Objeto == None:
156         sys.exit()

```

```

158     #self.Objeto = Objeto
159     Objeto = Objeto
160     self.nCI = len(Objeto.setVectores)
161     time = Objeto.time
162     self.bondad = Objeto.bondad
163     self.m_r = Objeto.M_R
164     #self.id_CI = Objeto.id_mejor_CI
165     self.dibujAutom()
166
167     @cronometro
168     def dibujAutom(self):
169         print self.bondad
170         #ID = self.id_CI
171         for i in xrange(self.nCI):
172             c = (i)#ID) # tupla que dara el numero de la regla y de la CI para guardar la
173             imagen.
174             matrizEstados = self.m_r[i]
175             pylab.matshow(matrizEstados,cmap='binary')
176             pylab.title("Celdas")
177             pylab.ylabel("Tiempo")
178             pylab.savefig("Test_Regla_CI%i.eps" %c)
179             pylab.close()
180             # Aqui salvamos el automata seleccionado
181             f = open('automata_CI_%i.txt' %c, 'w')
182             fout = csv.writer(f, delimiter = '\t')
183             fout.writerow(matrizEstados)
184             f.close()
185
186     """Esta parte del codigo sirve para que el programa se ejecute solo si el modulo
187     es llamado como programa y no al importarlo."""
188
189     if __name__ == '__main__':
190         #print "¿Tienes reglas que importar? ([y]/n):"
191         #respuesta = raw_input()
192
193         #if respuesta != 'n':
194             #print "Escribe el nombre del archivo con la extension."
195             #archivo = raw_input()
196             archivo = 'mejor.txt'
197             reglasImportadas = open(archivo, 'r')
198             setreglas = reglasImportadas.readlines()
199             nreglas = len(setreglas)
200             listaReglas = []
201             regla = setreglas[0].split()
202             for i in xrange(len(regla)):
203                 regla[i]=int(regla[i])
204
205             automata = Automata(initReglas = regla)
206         # else:
207         #     automata = Automata()
208
209         AutomImagen(automata)

```



## A.3. Filtro

```

1  #!/usr/bin/env python
2  # -*- encoding: utf-8 -*-
3  from matplotlib import pylab
4  """ Autor: Daniel Lopez Coto """
5
6  archivo = 'nombre_automata.txt'
7  f = open(archivo, 'r')
8  filas = f.readlines()
9  nfilas = len(filas)
10 m = [] # Matriz del automata
11 for i in xrange(nfilas):
12     fila = filas[i].split()
13     longfila=len(fila)
14     for j in xrange(longfila):
15         fila[j] = int(float(fila[j]))
16     m.append(fila)
17
18 print ("¿Predomina el blanco? ([y]/n)")
19 resp = raw_input()
20 m_new = []
21 new_fila = []
22 for r in xrange(nfilas):
23     n_elem = len(m[r])
24     fila_actual = m[r][:]
25     elto = fila_actual[:]
26     if resp != "y":
27         for l in xrange(len(elto)):
28             if elto[l] == 1:
29                 elto[l] = 0
30             else:
31                 elto[l] = 1
32     new_elto = elto[:]
33     for i in xrange(-3,n_elem-3):
34         izq_suma = elto[i-3] + elto[i-2] + elto[i-1]
35         der_suma = elto[i+1] + elto[i+2] + elto[i+3]
36         suma = izq_suma + der_suma
37         if suma <= 3 and elto[i] == 1:
38             new_elto[i] = 1
39         else:
40             new_elto[i] = 0
41     if resp != "y":
42         for l in xrange(len(elto)):
43             if new_elto[l] == 1:
44                 new_elto[l] = 0
45             else:
46                 new_elto[l] = 1
47     new_fila.append(new_elto)
48 m_new = new_fila[:]
49
50 pylab.matshow(m_new,cmap='binary')
51 pylab.title("Celdas")
52 pylab.ylabel("Tiempo")
53 pylab.savefig("Filtrado.eps")
54 pylab.show()

```

# Bibliografía

- [1] Bastien Chopard and Michel Droz. *Cellular Automata Modeling of Physical Systems*. Cambridge University Press (1998).
- [2] James P. Crutchfield, Melanie Mitchell. “*The Evolution of Emergent Computation*”. *Proceedings of the National Academy of Science USA*. 92:10742-10746 (1995).
- [3] Rajarshi Das, James P. Crutchfield, Melanie Mitchell, James E. Hanson. “*Evolving Globally Synchronized Cellular Automata*”. *Proceedings of the Sixth International Conference on Genetic Algorithms*, p.336-343. San Francisco CA, Ed. Morgan Kaufmann (1995).
- [4] Chris Langton *Computation at the Edge of Chaos: Phase Transitions and Emergent Computation*. Physica D, v.42, p. 12-37 (1990).
- [5] Norman H. Packard *Adaptation toward the edge of chaos*. In *Dynamic Patterns in Complex Systems*, p.293-301. World Scientific (1988).
- [6] Warren Weaver. “*Science and Complexity*”. Rockefeller Foundation, New York City. *American Scientist*, 36: 536 (1948).
- [7] Juan Ignacio Vázquez, Javier Oliver. “*Evolución de autómatas celulares utilizando algoritmos genéticos*”. Facultad de Ingeniería - ESIDE, Universidad de Deusto. 48007 Bilbao, Vizcaya-España (2008).
- [8] Hiroki Sayama. “*Introduction to the Modeling and Analysis of Complex Systems*”. Binghamton University, SUNY (2015).
- [9] Stephen Wolfram. *Statistical Mechanics of Cellular Automata*. *Reviews of Modern Physics*, v.35, no.3, p.601-644 (1983).
- [10] Stephen Wolfram. “*A New Kind of Science*”. Wolfram Media (2002).
- [11] Roberto Poli. “*A Note on the Difference Between Complicated and Complex Social Systems*”. Department of Sociology and Social Research, University of Trento; UNESCO Chair in Anticipatory Systems (2013).
- [12] Thomas Schelling. “*Dynamics Models of Segregation*”. Harvard University (1971).
- [13] Toffoli T and Margolus N. “*Cellular Automata Machines*”. The MIT Press Cambridge, Massachusetts (1987).