El Juego de la Vida: una máquina Universal de Turing

Berta Olano Guillén

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Director del trabajo: José Carlos Ciria Cosculluela

Junio de 2024

Summary

The Game of Life, developed by the British mathematician John Conway in 1970 appears to be a simple game: a two-dimensional grid of cells, either dead of alive, evolving over time according to a few rules.

The aim of this project is to show that from such apparent simplicity emerges a complex behaviour with surprising computational properties. The purpose is to explore the ability to act as a Turing Machine. A Turing Machine, proposed by Alan Turing in 1936 is a theoretical model of computation that captures the notion of algorithm.

In order to explore the game we consider increasing abstraction levels. The fist level of abstraction consists in the description of basic patterns of cells and how they evolve in time. At a higher level of abstraction, we can implement logical gates which form the basis of Boolean logic and let us build more complex circuits capable of performing arithmetic operations, memory storage and other computational tasks. Finally, we will break down a Turing Machine into modules, each performing a specific task.

Índice general

Capítulo 1

Introducción

Al presentar al público general el Juego de la Vida, Martin Gardner escribía: *"La mayor parte del trabajo de John Horton Conway [...] se ha centrado en matemática pura [...]. Además de su trabajo serio, Conway también se divierte con la matemática recreativa"* [5].

Entendemos, más aún tratándose de M. Gardner, que oponer matemática recreativa a matemática seria es un recurso humorístico que no tiene una connotación despectiva. Los Juegos matemáticos son un elemento de motivación al estímulo intelectual. Además, de la aparente simplicidad de su planteamiento pueden emerger comportamientos y propiedades complejas. Un buen Juego puede estimular y desarrollar la intuición. El ajedrez y el go son aprendizajes de estrategia; la teoría de Juegos modeliza las interacciones entre agentes inteligentes que compiten en busca de su propio beneficio (el dilema del prisionero ha sido objeto de publicaciones en Science, y la Enciclopedia de Filosofía de Stanford le dedica una veintena larga de páginas).

El Juego de la Vida ilustra cómo, partiendo de dos reglas muy sencillas, emerge un comportamiento complejo. Las reglas están inspiradas en las condiciones que necesitan los seres vivos para prosperar: tanto el aislamiento como la superpoblación conducen a un desastre demográfico, y es necesaria una masa crítica para garantizar la viabilidad de una nueva generación. En particular, en este trabajo nos centraremos en argumentar si el Juego es Turing-completo: permite diseñar cualquier máquina de Turing.

Empezaremos definiendo formalmente el Juego de la Vida, las máquinas de Turing y las máquinas contadoras (también conocidas como máquinas de registro de Minsky).

En las dos siguientes secciones resumiremos dos argumentaciones en defensa de la universalidad del Juego de la Vida: la sugerida por el propio Conway en [2] y la propuesta en [9]. La primera ilustra cómo implementar los elementos fundamentales de la arquitectura de von Neumann: la memoria (almacenando información mediante la posición de configuraciones específicas llamadas bloques) y el procesamiento (implementando bits, flujos de bits y puertas lógicas). La segunda diseña una máquina de Turing, con su función de transición y su cinta. Para dar un soporte visual que facilite la comprensión de estos elementos, hemos creado un vídeo https://youtu.be/imtFCaQL2q0 al que se hará referencia en algunos puntos de este trabajo.

Terminaremos haciendo una breve reflexión sobre la robustez del Juego: ¿de qué forma podemos flexibilizar sus condiciones de modo que siga siendo Turing-completo?

Sin embargo, tras estudiar en detalle estos argumentos surge la pregunta ¿es esto suficiente para probar la universalidad del Juego de la Vida? En todos los artículos explorados a lo largo de este trabajo y en multitud de páginas web dedicadas al tema, se asume que esta propiedad se cumple dando para ello argumentos que soportan esta hipótesis. Aún así, no existe hasta la fecha una prueba rigurosa capaz de demostrar que es posible construir una máquina Turing-universal utilizando el Juego de la Vida. Abordaremos, por tanto, algunas de las condiciones necesarias que apuntan a que esto es posible.

Capítulo 2

Primeras definiciones

2.1. Autómatas celulares

Un autómata celular es un modelo matemático simple basado en una matriz de células ordenadas. Estas células poseen una única propiedad, que puede variar con el tiempo, a la que llamaremos estado. El conjunto de valores que puede tomar ese estado debe ser finito y cambia según unas reglas de transición. Estas reglas indican a qué estado pasará una célula según el estado actual de las células vecinas. Habitualmente, se aplican a todas las células al mismo tiempo, formando períodos discretos de tiempo a los que llamamos generaciones.

Definición 1. Utilizando como referencia el artículo [6] podemos dar una definición formal de autómata celular. Un autómata celular A se puede expresar como una 5-tupla $(\mathbb{Z}^d, Q, N, f, q_0)$, donde:

- $\mathbb{Z}$ es el conjunto de los enteros y $\mathbb{Z}^d = \{(a_1, ..., a_d) \mid a_i \in \mathbb{Z} \text{ para } i = 1, ..., n\}$ (d>0).
- Q es el conjunto finito de estados en que puede estar cada célula.
- N es el índice de vecindad que define las células vecinas de cada célula en la disposición regular. Es decir, $N = \{n_i \mid n_i \in \mathbb{Z}^d, i \in \mathbb{N}\}$ tal que para una celda $a \in \mathbb{Z}^d$, cada celda en $\{(a + n_1), ..., (a + n_n)\}$ son sus celdas vecinas. Además, se denota como n a la cardinalidad de N.
- Una función de transición, f, que será $f : Q^{n+1} \to Q$ si es un autómata determinista, es decir, dado el estado de una celda y sus vecinas, siempre corresponde el mismo estado o $f : Q^{n+1} \to 2^{\#Q}$ si es un autómata no determinista. En este último caso, dado el estado de una celda y sus vecinas, f hace corresponder un vector de tantas componentes como el cardinal de Q que tendrán valor 1 si y solo si se puede llegar a ese estado.
- $q_0 \in Q$ es el estado quiescente, que cumple $f(q0, ..., q0) = q0$.

Además, Wolfram [11] clasifica los autómatas celulares en cuatro clases, según su comportamiento a partir de una configuración inicial aleatoria. La primera clase está formada por los autómatas con comportamientos uniformes, esto es, si casi todas las condiciones iniciales llevan al mismo estado final. La segunda clase son los que tienen comportamientos cíclicos aislados, o sea, existen varios estados finales formados por estructuras que se repiten cada pocas generaciones o se mantienen siempre igual. La tercera clase son los autómatas con comportamientos caóticos en los que se siguen formando estructuras a través de los diferentes estados. La última clase está formada por aquellos autómatas con comportamientos complejos, combinan aleatoriedad y orden y las estructuras son capaces de moverse e interactuar modelando comportamientos complejos.

En este caso, vamos a estudiar más en detalle el Juego de la Vida, también conocido como Game of Life (GoL). Un ejemplo de autómata celular determinista desarrollado por John Conway

y que fue popularizado alrededor de 1970. El Juego de la Vida es un autómata de clase IV, según la clasificación de Wolfram [11], es decir, se generan estructuras que por sí mismas son sencillas, pero que evolucionan e interactúan en formas muy complicadas. Las células tienen dos posibles estados: viva o muerta. Y solo hay dos reglas que definen el cambio de estado entre generaciones:

- Si una célula viva tiene dos o tres vecinos vivos seguirá viva en la siguiente generación. En cualquier otro caso, la célula morirá.
- Si una célula muerta tiene exactamente tres células vecinas vivas, en la siguiente generación esta célula vivirá.

Llamamos células vecinas a aquellas que están en contacto directo: cada una de las ocho células que la tocan (arriba, abajo, derecha, izquierda y las cuatro diagonales). Entonces, utilizando la última definición, podemos ver el Juego de la Vida como una particularización de un autómata celular.

Definición 2. Definimos el Juego de La Vida como una 5-tupla (Z^2, Q, N, f, q_0), donde:

- $Q = \{0, 1\}$ es el conjunto de estados donde 0 representa una célula muerta y 1 representa una célula viva.
- N es la vecindad de Moore, donde cada célula tiene 8 células vecinas (las ocho adyacentes). En concreto, $N = \{(n_x, n_y) \mid n_x, n_y \in \{-1, 0, 1\}\} \setminus (0, 0)$, esto es,
  $N = \{(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)\}$.
- $f : Q^9 \to Q$ es la función de transición que sigue las reglas del Juego de la Vida de Conway (definidas previamente), donde el nuevo estado de una célula depende del número de células vecinas vivas.
- $q_0 = 0$.

2.2. Máquina de Turing

La máquina de Turing es un concepto matemático que desarrolló Alan Turing en 1936 y representa un modelo teórico de un dispositivo que permite realizar cómputos. De manera intuitiva, podemos entender la máquina de Turing como una cinta infinita dividida en casillas, cada una de las cuales puede contener un símbolo perteneciente a un alfabeto finito. La máquina cuenta con un cabezal colocado en una de las casillas y que puede leer o escribir símbolos y moverse a izquierda o derecha basándose en unas reglas previamente definidas. También cuenta con una función de transición que tiene como entradas dos enteros (i, j) y devuelve tres enteros (k, l, D), con $D \in \{0, 1\}$, correspondientes a la instrucción $q_i, S_j \to q_k, S_l, \{L, R\}$ donde q_i es el estado actual, S_j el símbolo de entrada, q_k el nuevo estado, S_l el símbolo de salida y L moverse a izquierda y R a derecha.

Definición 3. Una máquina de Turing se define mediante una 3-tupla (Γ, Q, δ) [1] donde Γ representa un alfabeto (un conjunto finito de símbolos), Q representa el conjunto finito de estados y δ es la función de transición:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\} \tag{2.1}$$

En cada paso, la máquina de Turing lee el símbolo en la cinta bajo su cabeza, consulta la función de transición δ para determinar la acción a tomar, actualiza su estado y mueve su cabeza a la izquierda (L) o a la derecha (R). La máquina empieza en uno de los estados de Q al que llamaremos estado inicial. Puede haber uno o varios estados finitos, al llegar a los cuales la ejecución se detiene.

2.2.1. Ejemplo de una máquina de Turing

Una máquina de Turing es capaz de resolver un problema concreto. En este caso, vamos a construir una máquina que duplique los caracteres '1' en una cadena de caracteres formada por '0' y una secuencia de '1' contiguos. Para ello construimos el autómata de la figura 2.1 que representa el comportamiento de la cinta. Teniendo en cuenta que el cabezal de lectura comienza

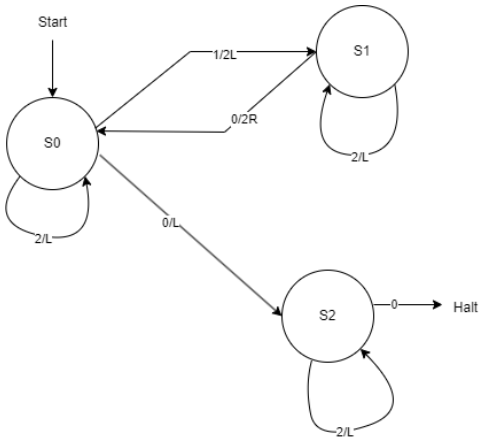

Figura 2.1: Autómata que modela la máquina de Turing capaz de duplicar los carácteres '1' en una cadena

en la posición del '1' más a la izquierda de la secuencia. En este autómata se pueden ver los tres estados, siendo S0 el de inicio. Se utiliza un alfabeto formado por 3 símbolos distintos: '0','1' y '2'. En las transiciones aparece el input para pasar al siguiente estado y el output, que, en este caso, es el movimiento que debe hacer el cabezal de lectura/escritura en la máquina. Desde el estado 2, si nos llega un input '0', nos vamos al estado de aceptación(Halt).

Podemos representar este comportamiento a través de una tabla de transición. Es decir, dado un estado actual y un input, nos da lugar a una salida como podemos ver en la tabla 2.1.

Estado actual	Input	Estado siguiente	Símbolo a escibir	Izquierda/Derecha
S0	0	S2	0	L
S0	1	S1	2	L
S0	2	S0	2	R
S1	0	S0	2	R
S1	2	S1	2	L
S2	0	halt	0	L
S2	2	S2	1	L

Cuadro 2.1: Tabla de transición

La salida de esta tabla se puede codificar en 1 Byte o equivalentemente 8 bits, que son dígitos que pueden tomar el valor 0 o 1. El último bit representa hacia que lado se mueve el cabezal de la cinta: 0 para izquierda y 1 para la derecha. Los tres siguientes para el símbolo a escribir: 000 para '0', 001 para 1 y 010 para '2' Los cuatro primeros para el estado siguiente: 0000 para '0', 0001 para 1 y 0010 para '2'.

De esta forma, modelamos el comportamiento de esta máquina de Turing, tal como se ve en la tabla 2.2. Así, por ejemplo, si utilizamos como entrada la cadena '000000111000000', nuestra máquina de Turing dará como resultado '000111111000000', ya que añade los nuevos '1' a la izquierda. Si ejecutamos el código mostrado en detalle en el anexo obtendremos el resultado de la figura 2.2.

(q_iS_j)	(q_iS_j) codificado	$(q_kS_l$ L,R) codificado	$(q_kS_l$ L,R)
(S0,0)	0000 000	0010 000 0	(S2,0,L)
(S0,1)	0000 001	0001 010 0	(S1,2,L)
(S0,2)	0000 010	0000 010 1	(S0,2,R)
(S1,0)	0001 000	0000 010 1	(S0,2,R)
(S1,2)	0001 010	0001 010 0	(S1,2,L)
(S2,0)	0010 000	halt 000 0	(halt,0,L)
(S2,2)	0010 010	0001 001 0	(S2,1,L)

Cuadro 2.2: Tabla de transición codificada

Figura 2.2: Estados inicial y final de la cinta a la que se aplica la máquina de Turing descrita en la tabla 2.1

2.3. Máquina contadora

Una máquina contadora es una máquina abstracta similar a una computadora utilizada como una herramienta matemática para explorar los límites de la computabilidad. Es equivalente a una máquina de Turing en su capacidad. La operación de una máquina contadora fue descrita por Minsky [7].

Definición 4. Consiste en un número finito de contadores (registros que almacenan enteros no negativos) controlados por un programa simple que consta de una lista de instrucciones etiquetadas. Estas instrucciones se ejecutan una tras otra al igual que en una computadora convencional, excepto que solo cuenta con tres instrucciones y los contadores teóricamente pueden contener cualquier número positivo, por grande que sea.

Un conjunto típico de intrucciones, donde *Id* es la etiqueta de esta instrucción, es:

- **id INC c next** incrementa el contador *c* y luego va a la instrucción etiquetada con *next*.
- **id DEC c next onZero** si el contador *c* es cero, va a la instrucción *onZero*; si no, decrementa el contador *c* y va a la instrucción *next*.
- **id HLT Halt** para

Un ejemplo que utiliza estas instrucciones es el siguiente, capaz de sumar los contenidos de c1 y c2, guardando el resultado en c2 y dejando c1 a 0:

01 DEC c1 02 03

02 INC c2 01

03 HLT

Capítulo 3

El Juego de la Vida

3.1. Diseño formal de patrones

Definición 5. Un patrón en el contexto del Juego de la Vida se refiere a una disposición específica de células vivas y muertas en una cuadrícula bidimensional. Un patrón puede ser tan simple como una sola célula o tan complejo como un patrón capaz de generar otros patrones.

Diseñamos un diagrama de clases siguiendo UML (Lenguaje Unificado de Modelado) para definir inequívocamente y de manera sistemática las clases necesarias que conformarán nuestro sistema. Aquí, definimos la clase patrón, que tiene el atributo nombre. Se distinguen dos tipos de

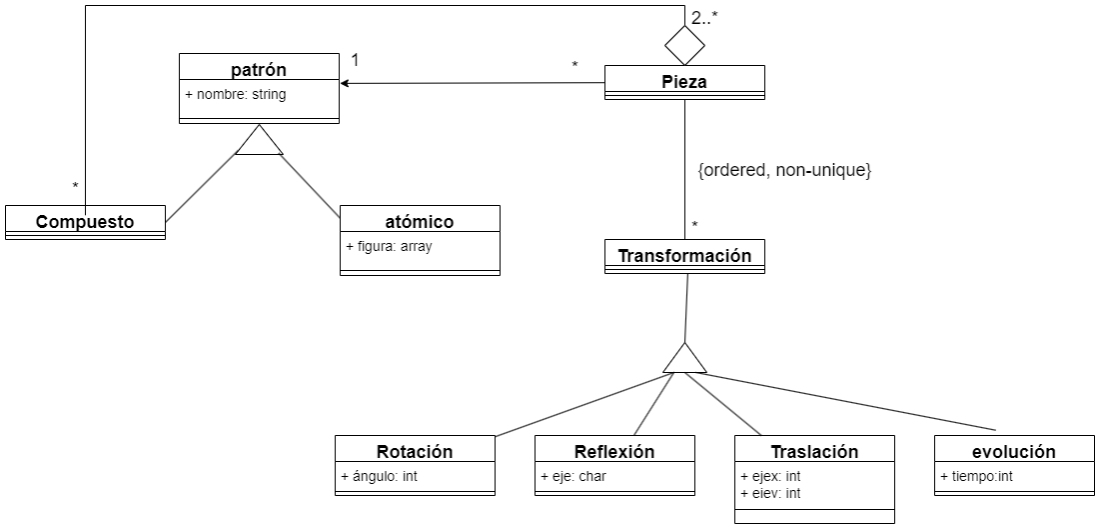

Figura 3.1: Diseño UML de clases

patrón: compuesto y atómico. Un patrón atómico tiene como atributo una figura, siendo esta una matriz de ceros y unos que lo caracterizan. Un patrón compuesto, sin embargo, es un conjunto de piezas. Cada pieza se obtiene a partir de un patrón que sufre una secuencia de transformaciones básicas. Estas transformaciones son rotación, reflexión, traslación y evolución, y dan lugar a diferentes estados de un mismo patrón. Las transformaciones están definidas con detalle en el anexo 8.1.

Este es un diseño independiente de la implementación, ya que, a la hora de programarlo, vamos a simplificar las clases que vamos a utilizar para que resulte más sencillo. De esta forma, tenemos una clase Pieza con los métodos de rotación, reflexión, traslación y evolución. Además tiene otros métodos necesarios para facilitar la implementación del Juego.

3.2. Patrones atómicos

A continuación, se muestran en la figura 3.2 algnuos ejemplos de patrones atómicos. Estos pa-

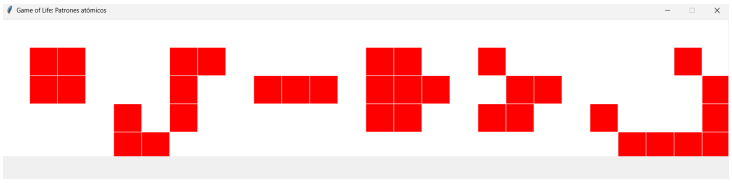

Figura 3.2: De izquierda a derecha: (a)Bloque (b)Eater o aniquilador (c)Segmento (d)Bloque ampliado (e)Glider (f)Nave espacial. El código se encuentra en el anexo 8.3.1

trones tienen distintos tipos de dinámica: El bloque (a) y el aniquilador (b) son estáticos: si están aislados, se mantienen invariantes en el tiempo. El segmento (c) tiene un comportamiento periódico con periodo T=2 (cada dos generaciones vuelve a su estado inicial). Tras siete generaciones, el bloque ampliado (d) deviene en cuatro segmentos. A partir de entonces el comportamiento es periódico con T=2.

El glider (planeador) (e) y la nave (f) se desplazan a lo largo del espacio. Van a ser esenciales en el desarrollo del trabajo ya que los vamos a utilizar como unidad lógica, es decir, una forma de representar un valor binario (bit). Definimos para ellos T como el periodo, o lo que es lo mismo, cada cuántas generaciones recuperan su forma original, y v como la velocidad a la que se desplazan. Conway denotó c, la velocidad de la luz, a la velocidad máxima a la que puede moverse un objeto (una casilla por generación). Expresaremos la velocidad de un patrón móvil en términos de c. Glider y nave tienen periodo T=4 (cada cuatro generaciones vuelven a su forma inicial). El glider se mueve en diagonal, a una velocidad de c/4. La nave se mueve en dirección horizontal a velocidad v=c/2. En la figura 3.3 se muestran las posiciones iniciales de una nave y un glider en un tiempo (gris) y doce generaciones más tarde (rojo):

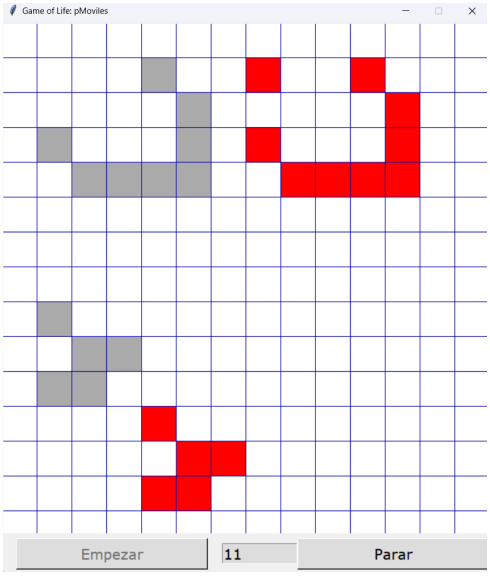

Figura 3.3: Glider y nave en las generaciones 0 y 11. Código en el anexo 8.3.2

3.3. Patrones Compuestos

3.3.1. Patrón generador: pistola

Este patrón "dispara" un glider cada 30 generaciones. En la figura se muestran la pistola en su estado inicial (a) y tras 60 generaciones (b): En la secuencia de la figura 3.4 podemos ver

Figura 3.4: (a)Pistola en la generación 0 (b) pistola en la generación 60. Código en el anexo 8.3.3

como lanza "balas" que, en realidad, es un flujo de gliders, cada 30 generaciones. Una pistola se construye a partir de los patrones bloque y bloqueAmpliado. En el minuto 00:00 del vídeo podemos observar cómo actúa este patrón en detalle.

3.3.2. Vaquero

El vaquero es un patrón de periodo T= 30. Consta de una figura central (abeja) cuyo movimiento es horizontal y está acotado por dos aniquiladores: al llegar a uno de ellos, rebota y cambia de sentido. El vaquero se construye a partir de los patrones bloque, bloqueAmpliado y

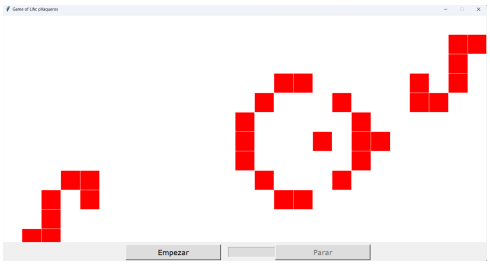

Figura 3.5: Vaquero

aniquilador (el código se muestra en 8.3.3). En el minuto 00:20 del vídeo podemos observar cómo actúa este patrón en detalle.

3.3.3. Definición de elementos

Pueden existir varios objetos que sigan un mismo patrón. Por ejemplo, puede haber varios gliders en distintas posiciones, orientaciones y formas (puesto que el periodo de un glider es T=4, puede adoptar cuatro formas distintas). Un objeto, pues, puede caracterizarse como un subconjunto $o \subseteq \mathbb{Z}^3$. Para cada elemento del subconjunto, las dos primeras componentes son los centros de las celdas con estado 1 en el instante que indica la tercera componente. Denotamos $o(t_n) \subseteq \mathbb{Z}^3$ a la figura que adopta el objeto en el instante t_n. El tiempo de Vida de un objeto es un intervalo de números naturales, correspondientes a los instantes de tiempo entre el nacimiento y la muerte del objeto; para un instante t_n fuera de ese intervalo, $o(t_n) = \varnothing$.

Definición 6. Sea F una secuencia de patrones que se desplazan, decimos que F es un flujo o tren si se cumple que $\forall g_1, g_2 \in F, \exists\ n_1, n_2 \in \mathbb{Z}$ tal que $g_1(t_{n_1}) = g_2(t_{n_1} + n_2 \times T)$ donde T es un número de generaciones que dependerá de qué patrón forme el flujo.

En concreto, si consideramos un flujo como una subsecuencia de los gliders generadores de una pistola, el valor de T será 30.

Definición 7. Un flujo o tren es denso si se cumple que $\forall g_1 \in F, \forall n_1, n_2 \in \mathbb{Z}\ \exists\ g_2 \in F$ tal que $g_1(t_{n_1}) = g_2(t_{n_2})$.

Además, podemos caracterizar un flujo mediante un objeto g_0, un tiempo inicial t_0 y una secuencia de bits tal que $(g_0(t_0), \{b_i\}) = \{g_i$ tal que $g_i(30 \times i) = g_0$ si $(i) = 1\}$.

Notar que $g_0(t_0)$ solo pertenece al flujo si $b_0 = 1$.

3.4. Dinámica entre patrones

Como hemos visto, podemos utilizar patrones que mantienen su forma a lo largo de las generaciones para modelizar conjuntos de datos. Si utilizamos un flujo de gliders para representar los bits, tendremos que en una sucesión la presencia de un glider representa el valor '1' y la ausencia representa el valor '0'. Además, necesitamos ser capaces de eliminar objetos, es decir, transofrmar el valor '1' en '0' y modificar trayectorias para modelar comportamientos lógicos. A continuación, veremos que ambas tareas pueden realizarse mediante colisiones.

3.4.1. Eliminar objetos

Hay dos formas básicas de eliminar un patrón:

- Destrucción de un objeto: es capaz de eliminar un solo glider sin dejar rastro (no hay células vivas alrededor). Se hace, por ejemplo, mediante una colisión entre dos gliders o una colisión entre el glider y un eater.
- Destrucción de varios objetos: se produce mediante la colisión de dos gliders, pero de forma que deja ciertas células vivas alrededor que durarán varias generaciones. Este es el caso, por ejemplo, de las colisiones que son capaces de generar un hueco de 8 gliders en un flujo.

Ambas se muestran en la figura 3.6(b). En el minuto 01:12 del vídeo podemos ver el funcionamiento de estas colisiones.

3.4.2. Desviar trayectorias

La figura 3.6(a) muestra dos ejemplos de colisiones cuyo resultado es desviar la trayectoria de un objeto. Los gliders de arriba colisionan y el situado más abajo hace un giro de 180º. A continuación, colisiona con un vaquero que lo desvía 90º. Precisamente el nombre de vaquero se aplica a patrones cuya funcionalidad es redirigir un objeto. En el minuto 00:36 del vídeo podemos ver el funcionamiento de estas colisiones.

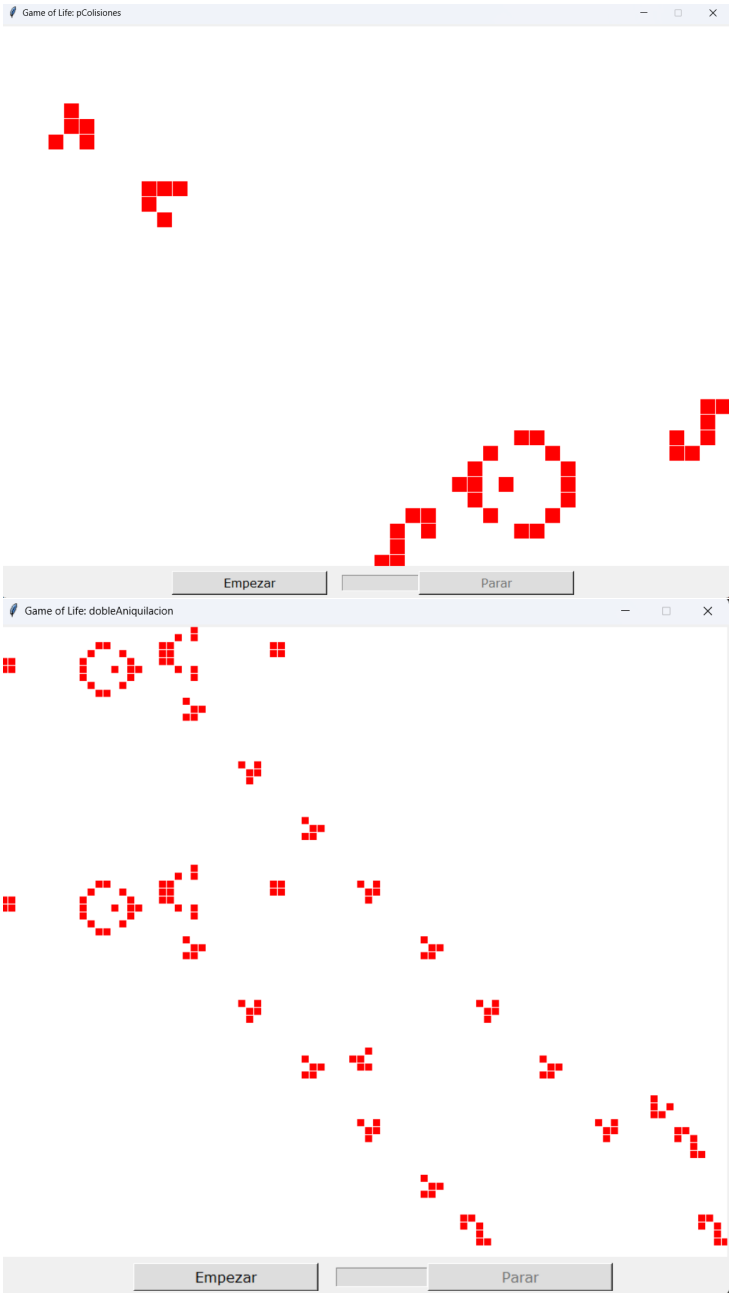

Figura 3.6: (a)Colisión entre dos gliders desviando uno de ellos 180º y haciendo desaparecer el otro. El glider desviado colisiona con el vaquero, que lo desvía 90º. (b)Dos ejemplos de colisiones glider-flujo, en la de más arriba se eliminan 8 gliders del flujo y en la de más abajo la colisión elimina un solo glider del flujo. Además, la colisión de los gliders del flujo con el aniquilador los elimina. Código en el anexo 8.3.4

Capítulo 4

El argumento de Conway

Para argumentar la universalidad del Juego de la Vida, John Conway razona que el autómata proporciona todos los elementos necesarios para diseñar un ordenador [3]. Hemos visto que es posible crear flujos de bits, generando gliders con una pistola y huecos mediante colisiones: la presencia de gliders modelaría el 1 y su ausencia el 0. En este capítulo veremos que, además, es posible:

- Implementar puertas lógicas (NOT, AND) y, a partir de ellas, un circuito lógico. Es posible redirigir el flujo de bits a través del circuito, por ejemplo, mediante colisiones con vaqueros.
- Implementar un registro de memoria persistente (estable en el tiempo). El registro consiste en una serie de direcciones de memoria en cada una de las cuales hay un bloque (que, recordemos, es un patrón estacionario). La posición del bloque codifica el valor almacenado en la dirección de memoria. Para cambiar dicho valor, basta desplazar el bloque mediante colisiones.

4.1. Componentes lógicos

Como hemos visto, podemos utilizar patrones que mantienen su forma a lo largo de las generaciones para modelizar conjuntos de datos. Si utilizamos un flujo de gliders para representar los bits, tendremos que en una sucesión la presencia de un glider representa el valor '1' y la ausencia representa el valor '0'.

Para poder utlizar estos valores necesitamos ser capaces de modificarlos según las necesidades lógicas del modelo que queramos representar. Para ello, utilizaremos las colisiones que, como se ha detallado anteriormente, nos permiten eliminar objetos y desviar trayectorias.

4.2. Construcción de una puerta NAND

Una vez que sabemos como representar cualquier señal binaria a través de un flujo de gliders o naves espaciales, si somos capaces de construir una puerta NAND, podremos modelar cualquier circuito lógico. A continuación, exploramos en detalle las componentes necesarias para hacerlo. Para construir una puerta NOT, es decir, una puerta que dé como resultado la palabra binaria de entrada negada, es necesario colocar una pistola que cree un flujo constante de gliders a 90º respecto del flujo de entrada. Si la colocamos de la manera adecuada, cuando un glider del flujo de entrada se encuentre con el flujo creado por la pistola, se producirá una colisión, dejando un hueco en el flujo creado por la pistola.

Así, si en el flujo de entrada hay un glider (representa el valor '1'), la colisión con la pistola generará una ausencia de glider en el flujo (representa el valor '0'). Si, por el contrario, en el flujo de entrada hay una ausencia de glider (representa el valor '0'), el flujo creado por la pistola seguirá su rumbo sin ser colisionado y, como resultado, tendremos un glider en el flujo (representa

el valor '1'). Esto es exactamente la función de una puerta NOT, negar un conjunto de datos de entrada.

Definición 8. Una puerta NOT está formada por un flujo denso acotado, generado por una pistola, A y un flujo B, tal que para el último glider de B, llamémoslo g_B, existe en A un glider g_A tal que verifica que $g_B = Traslacion(ReflexionEjeY(g_A), 1, 30n + 3)$ con $n > 0$.

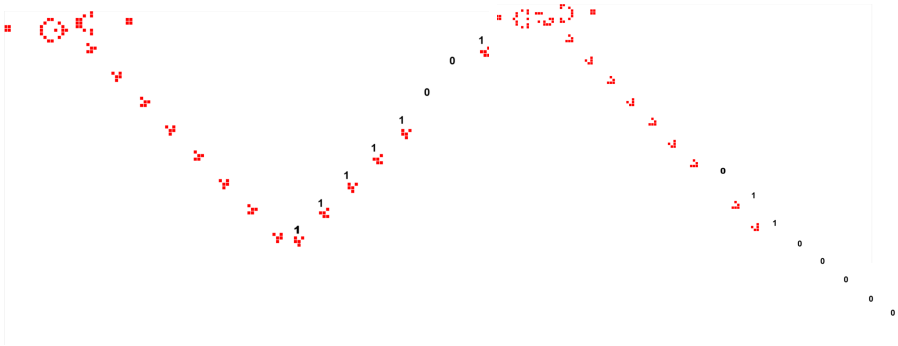

Figura 4.1: Puerta NOT, niega el flujo de entrada a través de una pistola. Código en el anexo 8.3.5 y en el minuto 01:51 del vídeo podemos ver el funcionamiento de esta puerta lógica.

Una vez que sabemos como negar un flujo es fácil ver cómo construir una puerta AND. Para ello, necesitamos colocar los dos flujos de entrada paralelamente y de nuevo una pistola que cree un flujo constante de gliders a 90º. Si en el primer flujo hay un glider (valor '1'), colisionará con el flujo de la pistola creando una ausencia de glider (valor '0'). Esta ausencia de glider no provocará una colisión con el segundo flujo. Así, si en el segundo flujo hay un glider, seguirá su camino (el valor final es '1') y si no lo hay se mantendrá ese hueco (valor '0'). Si por el contrario no hay glider en el primer flujo, no se producirá una colisión y el flujo de la pistola seguirá su camino. Ese glider colisionará con el segundo flujo, si es que existe, y si hay ausencia de glider no producirá tampoco un glider en la salida.

En resumen, solo se producirá un glider en el flujo de salida si en ambos flujos de entrada hay un glider en la misma posición. Este es exactamente el comportamiento de una puerta AND. Como se puede ver en la figura 4.2, es necesario colocar un eater para parar el flujo de la pistola

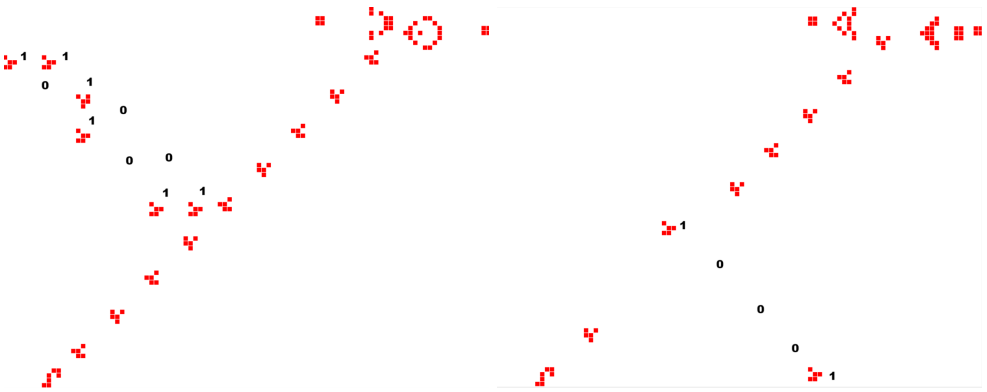

Figura 4.2: Puerta AND, produce un glider en la salida si y solo si hay glider en ambos flujos de entrada. Código en el anexo 8.3.6 y en el minuto 02:43 del vídeo podemos ver el funcionamiento de esta puerta lógica.

que resulta si hay ausencia de gliders en ambos flujos de entrada y, por tanto, no se produce ninguna colisión que elimine esos gliders.

Definición 9. Una puerta AND está formada por dos flujos de entrada A y B y un flujo denso acotado, en general generado por una pistola, verificando:

1. Los flujos de entrada están situados a la misma altura en la cuadrícula y a una distancia horizontal múltiplo de 8.

2. Cada flujo de entrada produce una puerta NOT con el flujo denso acotado.

Para construir una puerta NAND es necesario invertir ambas entradas con una puerta NOT, como acabamos de ver, y utilizarlas como entradas para una puerta AND. La salida de esa puerta negada representará la salida de una puerta NAND. El comportamiento de esta es: se produce un glider en el flujo de salida para cualquier combinación en los flujos de entrada, salvo en el caso de que haya un glider en ambos. Aunque no vamos a entrar en detalle con la demostración,

A	**B**	**Salida**
0	0	1
0	1	1
1	0	1
1	1	0

Cuadro 4.1: Tabla de verdad de una puerta NAND

se puede ver que las puertas NAND son universalmente completas, esto es, se puede construir cualquier circuito lógico a través de ellas. Por tanto, con esto demostramos que efectivamente podemos modelar cualquier circuito lógico usando patrones del Juego de la Vida.

4.3. Construcción de bloques de memoria

Una vez que hemos probado que podemos construir componentes lógicos a través del Juego de la Vida, esto es, somos capaces de construir una unidad aritmético-lógica (ALU), solo nos falta demostrar que también podemos almacenar información. Con el concurso de estos dos elementos, Conway argumentó que era posible construir un ordenador a través de patrones del Juego de la Vida.

En cuanto a la unidad de memoria externa, debe poder almacenar números arbitrariamente grandes. Para construirla, contaremos con registros que contienen cada uno un bloque, cuya distancia a la computadora (en una cierta escala) indica el número que contiene. Así, para incrementar o decrementar el número contenido en un registro, será necesario desplazar el bloque más cerca o más lejos de un determinado punto de la cuadrícula. Tenemos tres instrucciones diferentes:

- Incrementar: aumenta en una unidad el contenido del registro.
- Decrementar: disminuye en una unidad el contenido del registro.
- Test: comprueba si el contenido del registro es 0 (el bloque está dentro de la computadora).

4.3.1. Incrementador

Para incrementar un registro en una unidad, debemos conseguir mover un bloque en diagonal 3 casillas en dirección noroeste (quedará 3 casillas más arriba y 3 más a la izquierda). Para ello, utilizamos dos gliders colocados de la forma que se puede ver en la figura 4.3. En el minuto 03:33 del vídeo podemos ver el funcionamiento de estas colisiones.

4.3.2. Decrementador

Para decrementar un registro en una unidad, es decir, mover el bloque en diagonal una casilla en dirección sureste, Conway propone una secuencia de tres colisiones como las mostradas en la

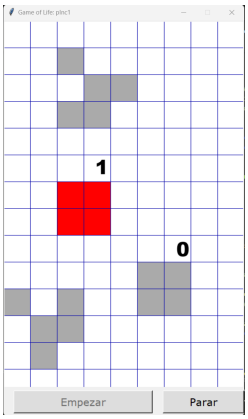

Figura 4.3: Desplazamiento de 3 casillas en diagonal en dirección NW

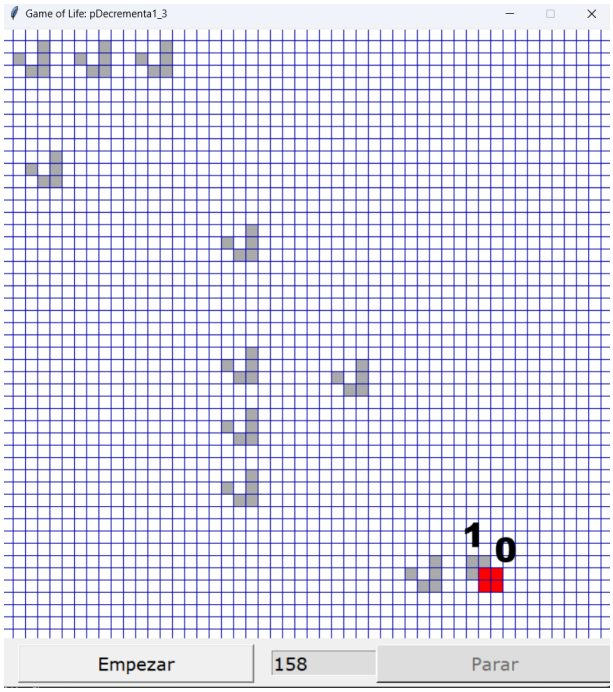

Figura 4.4: Desplazamiento de 1 casilla en diagonal en dirección SE

figura 4.4, cada una de las cuales desplaza el bloque diagonalmente una casilla. Son necesarias 160 generaciones para generar este movimiento. En el minuto 03:43 del vídeo podemos ver el funcionamiento de estas colisiones.

Capítulo 5

El Argumento de Rendell

Como hemos visto en los capítulos anteriores, el Juego de la Vida es un autómata celular capaz de simular una amplia variedad de sistemas computacionales. Como prueba de concepto, Paul Rendell [9] diseñó una máquina de Turing. En este capítulo exploraremos en detalle la implementación de algunos de sus componentes, ilustrando cómo los patrones que hemos descrito permiten la construcción de una máquina de Turing, tal y como se puede ver en la figura 5.1.

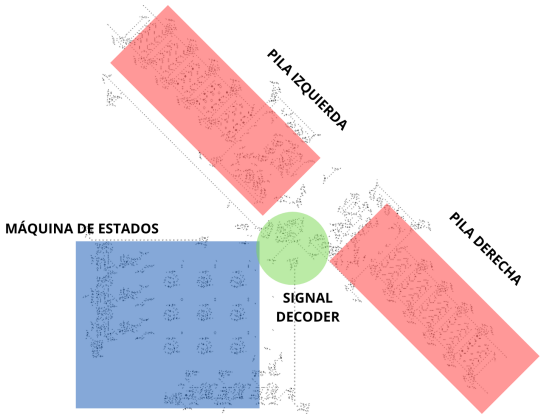

Figura 5.1: patrones que forman una máquina de Turing

El núcleo de esta demostración es la capacidad de construir diversos componentes, como una máquina de estados y una cinta semi-infinita en el marco del Juego de la Vida. Este análisis nos permitirá comprender parte de la complejidad del diseño de Rendell.

5.1. Idea de construcción

Para poder construir esta máquina, son necesarios varios componentes que primero estudiaremos a un nivel alto de abstracción para dar una intuición de cómo funcionan. Necesitaremos implementar una tabla de transición y dos pilas que actuarán como una cinta.

5.1.1. Máquina de estados

La máquina de estados implementa una función de transición, que tiene como entradas dos enteros (i, j). Devuelve tres enteros (k, l, D), con $D \in \{0, 1\}$, correspondientes a la instrucción $q_i, S_j \to q_k, S_l, \{L, R\}$ ($D = 0, 1$ según si el cabezal de la cinta se tiene que desplazar a la izquierda o a la derecha).

5.1.2. Pilas

La cinta de una máquina de Turing es infinita en ambas direcciones. En su estado inicial todas las celdas, salvo un número finito de ellas, están en blanco [7]. Partiendo de una celda inicial, el cabezal de la máquina de Turing se desplaza bien a la derecha, bien a la izquierda. Una cinta infinita presenta una dificultad: no existe una celda especial (celda 0), a partir de la cual se almacene el contenido inicial de la cinta (por ejemplo, el código de una máquina de Turing con el que trabaja la máquina universal). En su lugar, consideramos dos semi-infinitas, una acotada a la izquierda (pR) y otra a la derecha (pL), de modo que el cabezal de la máquina de Turing apunta siempre a la primera celda de pR: pR contiene la celda a la que apunta el cabezal y todas las que hay a su derecha; pL contiene todas las celdas a la izquierda del cabezal. Esto es: consideramos que el cabezal está fijo y que es la cinta la que se desplaza. Para modelizar cada cinta usaremos una pila: una estructura de datos FIFO (first in first out). Podemos identificar las celdas de la pila mediante números enteros no negativos. Dada una pila p, denotamos p[i] al valor almacenado en la i-ésima celda; la primera celda, de índice 0, es donde se añade el elemento nuevo. La pila tiene dos operaciones básicas: apilar (push), en que se añade un elemento al inicio de la pila (la celda tiene índice 0) y desapilar (pop), en que se extrae el primer elemento. En particular:

```
funcion p.push (x)
i <- 0
    MIENTRAS p[i] != ''
        p[i+1] <- p[i]
        i <- i+1
    p[0] <- x
funcion p.pop()
    resultado <- p[0]
    i <- 1
    MIENTRAS p[i] != ''
        p[i-1] <- p[i]
        i <- i+1
    p[i-1] <- ''
    DEVUELVE resultado

```

De este modo, la parte de instrucción S_l, D (escribir S_l en la celda marcada y desplazarse en la cinta a derecha o izquierda según el valor de D) sería:

```
funcion ejecutaInstruccion(Sl,D)
    pR.pop()
    pR.push(Sl)
    SI D==1
        pL.push(pR.pop())
    SI NO
        pR.push(pL.pop())

```

El esquema general de construcción de la máquina de Turing tendrá una estructura como la que se puede ver en la figura 5.2.

Para ello, damos como prueba de concepto varios ejemplos de máquinas de Turing que se pueden construir siguiendo este esquema y que están explicadas en detalle en el anexo 8.2:

1. Duplicador de '1': es el ejemplo de máquina que se desarrolla en este trabajo y sobre el cual Rendell basó su argumento de universalidad.

2. Generación de la secuencia $\widehat{01}$ (representación binaria del número $\frac{1}{3}$). Es el primer ejemplo que desarrolla Turing en el artículo en el que propuso su máquina [10].

3. Generación de la secuencia '01011011101111...'. Es el segundo ejemplo que propone Turing con el objetivo de demostrar que existen números irracionales que son computables.

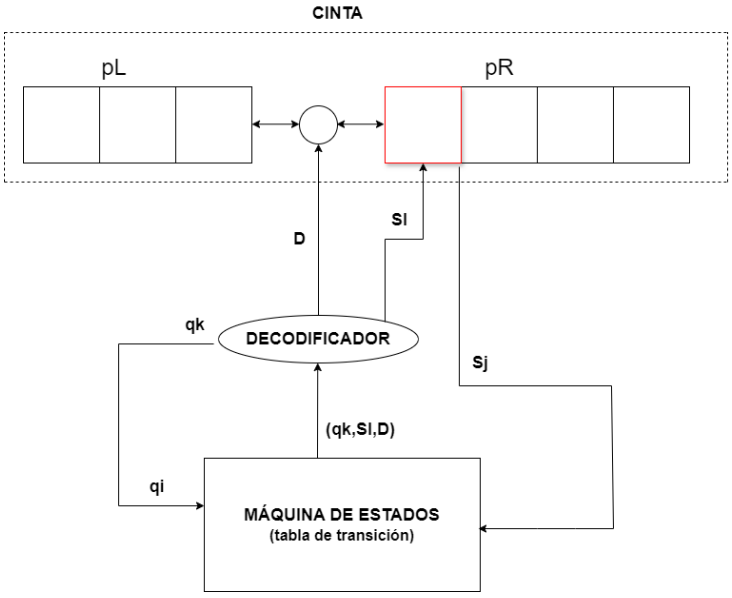

Figura 5.2: Idea intuitiva del diseño de una máquina de Turing

4. Suma y producto de números arbitrarios.

A continuación, se detalla el funcionamiento de uno de los componentes principales de la máquina: una celda de la máquina de estados. La limitación de extensión de este trabajo no permite desarrollar en detalle el resto de componentes. Nuestro objetivo es mostrar cómo utilizar los flujos de bits y colisiones para modelizar comportamientos. El resto de componentes funcionan de una manera muy similar.

5.2. Celda de la máquina de estados

El objetivo de la máquina de estados es implementar la función de transición que puede describirse mediante un conjunto de intrucciones del tipo:

$q_i, S_j \rightarrow q_k, S_l, \{L, R\}$

La entrada (q_i, S_j) se implementa mediante la estructura de la máquina: esta está formada por una matriz de celdas, con tantas filas como estados tiene la máquina de Turing y tantas columnas como símbolos contenga el lenguaje. En el ejemplo visto en la tabla 2.2, hay tres estados y tres símbolos. La máquina de estados constará de una matriz de 3x3 celdas.

La salida $(q_k, S_l, \{R/L\})$ se implementa en las distintas celdas. Todas ellas tienen la misma estructura, pero distinto contenido. El contenido de la celda [i,j] es un flujo de gliders que codifican los valores de k y l y la dirección de desplazamiento del cabezal (1 a la derecha, 0 a la izquierda). La estructura es una serie de patrones que generan periódicamente el flujo de gliders y los mantienen encerrados en el interior de la celda.

En nuestro ejemplo, el contenido de cada celda codifica una serie de ocho bits. Los cuatro primeros codifican el nuevo estado (q_i) y los tres siguientes el símbolo que se escribe al ejecutar la instrucción (S_l). El flujo de gliders se repite con un periodo T=240.

En la figura 5.3 mostramos dos celdas que contienen dos flujos de información diferente: '11111111' y '10011001' respectivamente. Los patrones que lo forman son una pistola (rojo) que crea el flujo inicial, otra pistola (azul) cuyo flujo llega al fanout (amarillo) que crea un flujo que reflejan 90^{o} los tres vaqueros (verde). En el minuto 03:53 del vídeo podemos ver el funcionamiento de una celda conteniendo '11111111' con más detalle.

Para ejecutar una instrucción, la máquina selecciona la celda correspondiente a la entrada ([i,j]) y genera una puerta de salida, de duración 240 generaciones, para que el contenido de la

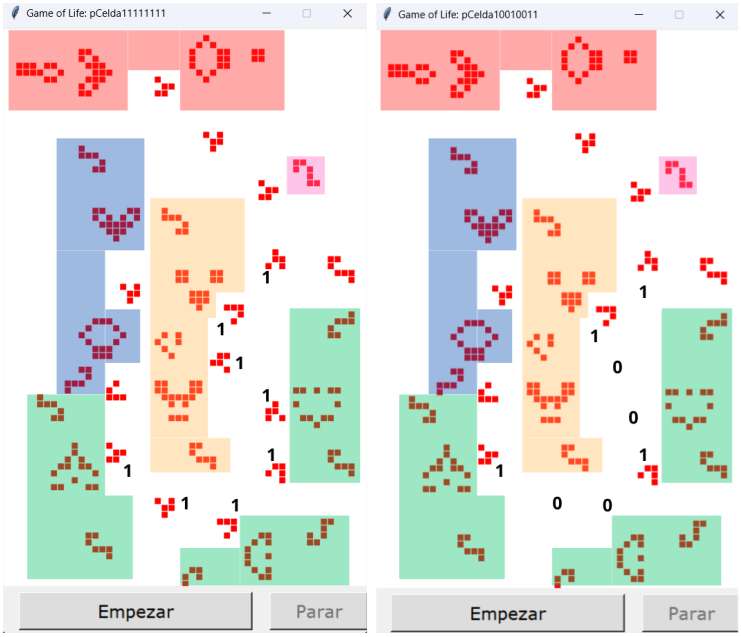

Figura 5.3: (a)Celda conteniendo '11111111' (b)Celda conteniendo '10010011'

celda salga al exterior. Al recibir la entrada (q_i, S_j), la máquina genera una nave espacial que se desplaza horizontalmente hacia la derecha sobre la i-ésima fila y otra que se desplaza hacia arriba a la derecha de la j-ésima columna. Ambas naves colisionan sobre la celda [i,j]. Los restos de la colisión interactúan con el Pentadecathlon (patrón situado en la esquina superior derecha en la figura 5.4), generando un glider que colisiona con el flujo de la pistola generando un hueco de 8 bits por el que escapa el contenido de la celda. En el minuto 04:51 del vídeo podemos ver

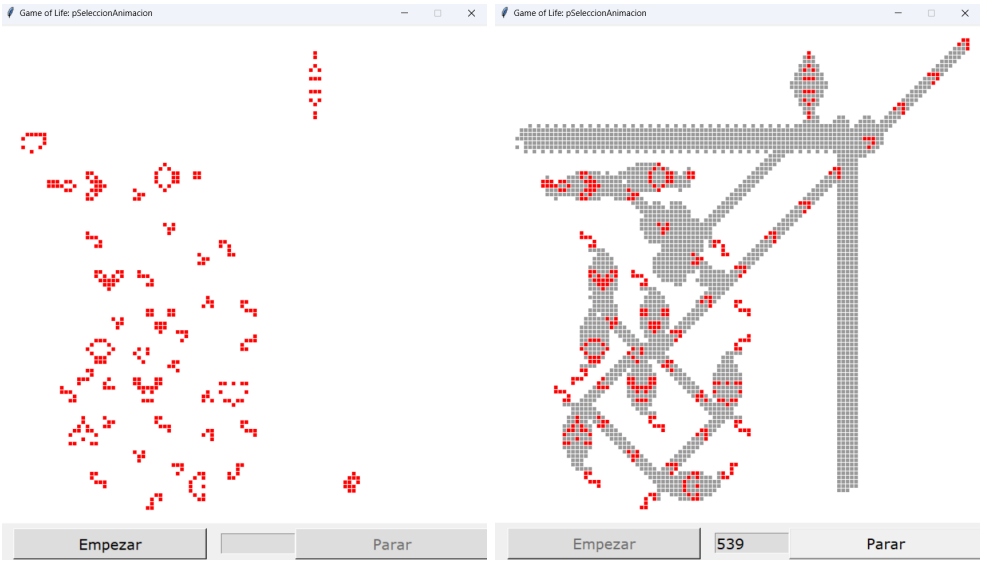

Figura 5.4: Celda que deja salir su contenido

cómo se selecciona una celda.

Capítulo 6

Robustez del Juego de la Vida

Pese a la sencillez de sus reglas, el Juego de la Vida exhibe un comportamiento complejo que lo encuadra en la clase IV de autómatas celulares definida por Wolfram. Esa complejidad lo hace candidato a ser Turing-universal: a poder construir a partir de él un ordenador de propósito general. Esta propiedad, para la cual no hemos encontrado una demostración rigurosa y formal, sigue siendo una conjetura verosímil. En esta sección, revisaremos la robustez del comportamiento del autómata: ¿en qué medida es posible generalizar sus propiedades de modo que mantenga un comportamiento complejo, susceptible de dar soporte a la computación universal?

Para que un autómata pueda dar soporte a la computación universal debe satisfacer dos requisitos necesarios: preservar la información para tiempos arbitrariamente largos (el resultado final de una computación está determinado por las condiciones iniciales) y permitir la propagación de la información a distancias arbitrariamente largas (condición necesaria para diseñar circuitos complejos, en que un flujo de bits debe atravesar múltiples puertas lógicas). Al considerar distintas variantes del Juego de la Vida, analizaremos si cumplen estas condiciones.

Una forma de analizar el comportamiento de un modelo con el tiempo es calcular su espectro de Fourier, S(f). Si $S(f) \propto \frac{1}{f^\alpha}$, con $0{,}5 < \alpha < 1{,}5$, se dice que el sistema tiene $\frac{1}{f}$-ruido. Esta propiedad es una evidencia de que el sistema recuerda su historia. Por tanto, la presencia de $\frac{1}{f}$-ruido es condición necesaria (aunque no suficiente) de que el sistema permite la computabilidad universal.

A lo largo de esta sección, denominaremos GL al modelo del Juego de la Vida tal como lo hemos considerado hasta ahora.

6.1. Juego de la Vida Estocástico

El Juego de la Vida Estocástico (Stochastic Game of Life, SGoL(p)) se define del siguiente modo: en cada generación, cada celda tiene una probabilidad p de actualizar su estado. Obviamente, SGL(1) coincide con GL y en SGL(0) no hay evolución. Partiendo de una configuración inicial aleatoria, para $0{,}9 < p < 1$ el sistema evoluciona a un estado final estático y de baja densidad (sobreviven muy pocas celdas). Para valores pequeños de p, el sistema converge a un estado estacionario con fluctuaciones aleatorias. En todo caso, ese estado final es muy robusto respecto de la densidad inicial: simulaciones con ocupaciones iniciales del $25\,\% - 99\,\%$ terminan convergiendo a estados finales indistinguibles [4]. El hecho de que, al avanzar el tiempo, el sistema pierda información del estado inicial, descarta el modelo SGL(p) (con p<1) para la computación universal.

6.2. Asincronía

El Juego de la Vida es un autómata celular síncrono: todas las celdas se actualizan simultáneamente en cada generación. Existen varias formas de relajar esa sincronía para obtener modelos

asíncronos. Los más apropiados para simular procesos reales biológicos son guiados por pasos o por tiempo.

En los métodos guiados por pasos (*step-driven methods*), a cada paso se actualiza una celda, el orden en que esto ocurre puede ser predefinido o aleatorio. En los métodos guiados por tiempo (*time-driven methods*), la probabilidad de que cada celda se actualice sigue una distribución exponencial creciente en el tiempo (inmediatamente después de una actualización, la celda tiende a mantener su nuevo estado; al avanzar el tiempo, la probabilidad de actualización crece exponencialmente).

En [6] se propone un modelo asíncrono, ASG, que permite una asincronía restringida: una celda puede estar desfasada como máximo un paso respecto de sus vecinas. Este simula exactamente el comportamiento del modelo GL.

6.3. Cambio de las reglas básicas

En el Juego de la Vida que propuso Conway el estado de una célula depende del estado anterior de las 8 células vecinas y de la propia célula. Ninagawa [8] propuso denotar $S_{xy}(t)$ al estado de la celda (x,y) en la generación t. Y este estado evoluciona según una función de transición F, tal que $S_{xy}(t+1) = F(S_{xy}(t), n)$ donde n es el número de células vecinas que tienen estado 1.

Siguiendo esta notación el Juego de la Vida original sigue las reglas:

- F(0,3) = 1
- F(1,2) = 1
- F(1,3) = 1
- F = 0 en cualquier otro caso

Cualquier autómata celular similar al Juego de la Vida, pero variando sus reglas se podrá escribir como:

- $F(0, n_i) = 1$ i=1,2,...,I
- $F(1, m_j) = 1$ j=1,2,...,J
- F = 0 en cualquier otro caso

Podemos usar la notación abreviada "$Bn_1n_2\cdots n_I/Sm_1m_2\cdots m_J$". Con esta notación, el Juego de la Vida original se denotaría B_3/S_{23}.

El objetivo es estudiar cómo varía la dinámica del Juego si variamos los valores en la función de transición, en concreto, vamos a examinar los resultados de F(0,2), F(0,3), F(0,4), F(1,1), F(1,2), F(1,3) y F(1,4). En total, hay $2^7 - 1 = 127$ posibles funciones de transición si excluimos en las que para cualquier valor F es 0.

Realizamos un análisis espectral de estos sistemas mediante la transformada de Fourier de una serie de estados $S_{xy}(t)$ de la celda (x,y) para t=0,1,...,T-1 que sigue la fórmula:

$$\hat{s}(f) = \frac{1}{T}\sum_{t=0}^{T-1} s_{xy}(t)\exp\left(-i\frac{2\pi t f}{T}\right) \quad (f = 0, 1, \ldots, T-1) \tag{6.1}$$

El espectro de potencia de Fourier se define como:

$$S(f) = \sum_{x,y} |\hat{s}_{xy}(f)|^2 \tag{6.2}$$

En este análisis, usaremos una cuadrícula de 120x120, condiciones de frontera absorbentes (nulas) bajo las cuales los sitios más allá de cada extremo se modifican para mantener el estado 0 y observaremos el resultado para T=4096 generaciones.

Casi todos los autómatas se pueden dividir en cuatro tipos distintos:

- Espectro de ruido blanco.
- Espectro de ruido blanco con picos en frecuencias particulares.
- Espectro lorentziano.
- Espectro lorentziano con picos.

Ejemplo de ruido blanco es la gráfica de B3/S123 que aparece en la figura 6.1 ya que contiene todas las frecuencias. Este tipo de ruido se caracteriza por gráficas planas.

En la gráfica de B3/S123 podemos observar un ruido lorentziano. Esto es, a bajas frecuencias S(f) es constante, pero a frecuencias más altas, $S(f) \propto \frac{1}{f^2}$. En concreto, podemos aproximar la gráfica con una función del tipo $\frac{A}{(f^{-n_1}+Bf^2)}$. Los parámetros resultantes son los que aparecen en la tabla 6.1.

Parámetro	Valor	Desviación estándar
A	9.170	3.4e-02
B	6.362e-06	8.2e-08
n_1	0.3399	9.6e-04

Cuadro 6.1: Valores de los parámetros y sus desviaciones estándar

La única que muestra la fluctuación del $\frac{1}{f}$-ruido es la del Juego de la Vida original (figura 6.1 (c)). La fluctuación $\frac{1}{f}$ es un proceso aleatorio en el que el espectro de potencia S(f) como función de la frecuencia f se comporta como $\frac{1}{f^\beta}$ con $0{,}5 \leq \beta \leq 1{,}5$ a una baja frecuencia.

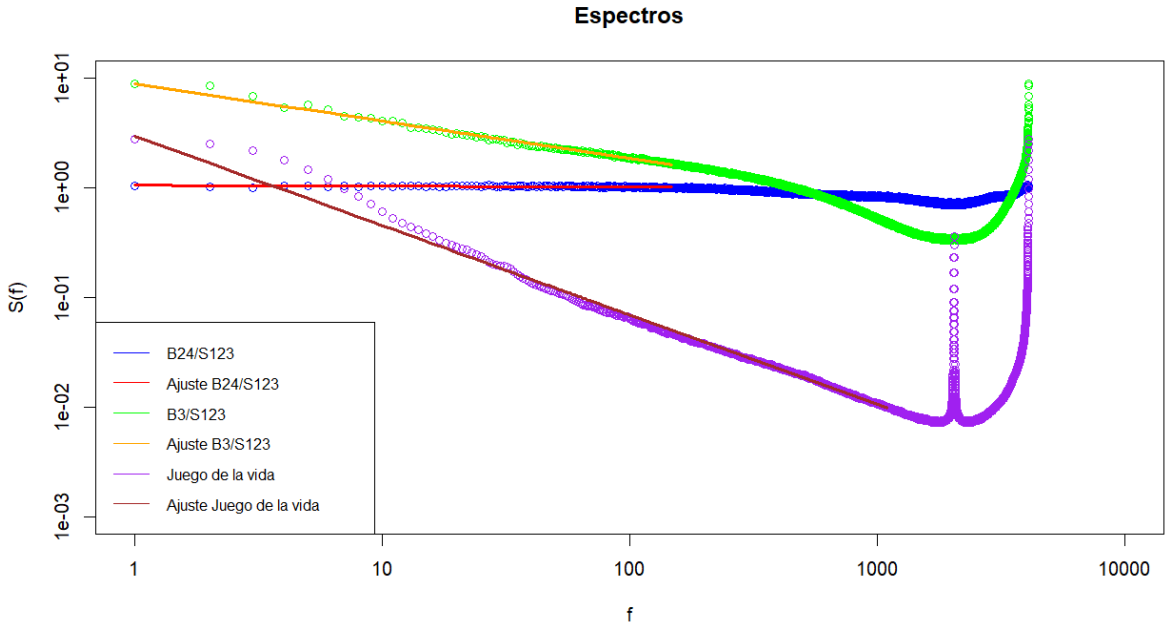

Figura 6.1: Espectros y sus ajustes para tres variaciones del Juego de la Vida

Procesando los valores obtenidos para estas tres combinaciones de reglas tenemos que $S(f) = \frac{1}{f^n} \implies \log S = -n \log f$. En concreto, obtenemos los valores de n y desviación estandar que se muestran en la tabla 6.2.

	n	**Desviación Estándar**
B24/S123	0.0079623	0.0009686
B3/S123	0.339333	0.002778
Juego de la Vida Original	0.816181	0.002129

Cuadro 6.2: Tabla con valores n y desviación estándar

La presencia de la fluctuación $\frac{1}{f}$ implica que el estado presente de un sistema está influenciado por el historial pasado del sistema o el estado presente del sistema tiene información de estados pasados. Por otro lado, la presencia de la universalidad computacional implica que un sistema tiene 3 cualidades: el sistema preserva información por periodos de tiempo arbitrariamente largos, el sistema permite propagar información por distancias arbitrariamente largas y el sistema debe ser capaz de procesar la información almacenada y transmitida.

Capítulo 7

Conclusiones

Durante este trabajo se ha abordado la construcción de una máquina de Turing utilizando el Juego de la Vida propuesto por Conway. A lo largo del desarrollo, se ha explorado la capacidad de este autómata celular para simular los componentes esenciales de una computadora.

Se han estudiado en detalle dos puntos de vista que tratan de demostrar que además el Juego de la Vida es capaz de realizar cualquier cálculo computacional, es decir, probar que es universal. El primero de ellos fue propuesto por John Conway, quien demostró que es posible crear puertas lógicas y, combinándolas, construir una unidad aritmético lógica. Así mismo, probó que existe una forma de almacenar datos en registros.

Por otro lado, el trabajo de Paul Rendell se centró en implementar una máquina de Turing concreta utilizando los patrones del Juego de la Vida. Aportó así una prueba de concepto que da soporte a la idea de que es posible contruir una máquina universal de Turing utilizando como único recurso el Juego de la Vida.

Por último, probamos que las reglas concretas propuestas por Conway aportan al Juego de la Vida una característica fundamental para su universalidad: la frecuencia sigue la fluctuación del $\frac{1}{f}$-ruido.

Sin embargo, a pesar de que estos argumentos suponen un avance signficativo para probar la universalidad computacional del Juego de la Vida y que todas las evidencias hasta la fecha apuntan a que sí posee esta propiedad, no existe todavía una demostración rigurosa y matemática que lo establezca de manera definitiva.

Durante este trabajo se ha contribuido al estudio de esta hipótesis y a la construcción y simulación de los componentes computacionales básicos de una máquina de Turing que permiten avanzar en la comprensión del Juego de la Vida como sistema universal de computación. Se han aportado pruebas sólidas que refuerzan la plausibilidad de esta propiedad, quedando pendiente una prueba formal.

Bibliografía

[1] Sanjeev Arora y Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] E. R. Berlekamp, J. H. Conway y R. K. Guy. «Winning ways for your mathematical plays, volume 4». En: *AK Peters/CRC Press* (2004).

[3] Elwyn R. Berlekamp, John H. Conway y Richard K. Guy. *Winning Ways for Your Mathematical Plays*. Vol. 2. Academic Press, 1982, págs. 915-924.

[4] Hendrik J Blok y Birger Bergersen. «Synchronous versus asynchronous updating in the "game of life"». En: *Physical Review E* 59.4 (1999), pág. 3876.

[5] Martin Gardner. *The fantastic combinations of John Conway's new solitaire game "life"*. Vol. 223. Scientific American, 1970, págs. 120-123.

[6] J. Lee et al. «Asynchronous game of life». En: *Physica D: Nonlinear Phenomena* 194.3-4 (2004), págs. 369-384.

[7] Marvin L. Minsky. *Computation*. Englewood Cliffs: Prentice-Hall, 1967.

[8] Satoshi Ninagawa, Masaki Yoneda y Satoshi Hirose. «1f fluctuation in the "Game of Life"». En: *Physica D: Nonlinear Phenomena* 118.1-2 (1998), págs. 49-52.

[9] Paul Rendell. *Turing machine universality of the game of life*. Cham, Switzerland: Springer International Publishing, 2016.

[10] Alan Mathison Turing et al. «On computable numbers, with an application to the Entscheidungsproblem». En: *J. of Math* 58.345-363 (1936), pág. 5.

[11] Stephen Wolfram. *New kind of science*. 1997.

Capítulo 8

Anexo

8.1. Definición formal de transformaciones

Para poder defininir los siguientes elementos, es necesario definir formalmente las diferentes transformaciones a las que podemos someter la matriz que representa un determinado patrón. Están basadas en el código necesario para implementarlas.

Definición 10. Sean una matriz de tamaño $m \times n$, $nFilas$ el número de filas a trasladar y $nCol$ el número de columnas a trasladar. La función de traslación $Traslacion : \mathbb{R}^{m\times n} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}^{Lx\times Ly}$ se define como:

$$Traslacion(\text{figura}, nFilas, nCol) = \text{figuraTraslada}$$

donde

$$\text{figuraTraslada}[y][x] = \begin{cases} \text{figura}[y - nFilas][x - nCol] & \text{si } nFilas \leq y < Ly \text{ y } nCol \leq x < Lx \\ 0 & \text{en otro caso} \end{cases},$$

con $Lx = m + nFilas$ y $Ly = n + nCol$.

Definición 11. Sea figura una matriz de tamaño $m \times n$. La función de reflexión respecto al eje Y $ReflexionEjeY : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ se define como:

$$ReflexionEjeY(\text{figura}) = \text{figuraReflejada}$$

donde $\text{figuraReflejada}[i][j] = \text{figura}[i][n - j - 1]$ para i que varía a lo largo de las filas y $j = 0$ hasta $n - 1$.

Definición 12. Sea figura una matriz de tamaño $m \times n$. La función de reflexión respecto al eje X $ReflexionEjeX : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ se define como:

$$ReflexionEjeX(\text{figura}) = \text{figuraReflejada}$$

donde $\text{figuraReflejada}[i][j] = \text{figura}[m - i - 1][j]$ para $i = 0$ hasta $m - 1$ y j que varía a lo largo de las columnas.

Definición 13. Sea figura una matriz de tamaño $m \times n$ y k el número de rotaciones.

La función de rotación $Rotacion : \mathbb{R}^{m\times n} \times \mathbb{Z} \to \mathbb{R}^{n\times m}$ se define como:

$$Rotacion(\text{figura}) = \text{figuraRotada}$$

donde $\text{figuraRotada}[i][j] = \text{figura}[j][n - i - 1]$ para i que varía de 0 a $n - 1$ y j que varía de 0 a $m - 1$. Esto realiza una rotación en sentido antihorario de 90^{o}, por tanto sería necesario repetirlo tantas veces como indique el parámetro k.

Definición 14. Sea una matriz que representa el estado inicial de un sistema.
La función de evolución $Evolucion : \mathbb{R}^{m\times n} \times \mathbb{Z} \to \mathbb{R}^{m'\times n'}$ se define como:

$$Evolucion(\text{figura}, nSteps) = \text{figuraFinal}$$

donde figuraFinal es el estado del sistema después de $nSteps$ pasos de evolución, definido por el siguiente proceso:

1. Extensión de la matriz original con ceros en cada borde:

$$\text{figuraExtendida} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \text{figura} & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

2. Aplicación de la reglas de evolución que actualizan el estado de la matriz extendida de acuerdo con las reglas del Juego de la Vida, previamente definidas.

3. Recorte de los ceros adicionales alrededor de la matriz actualizada:

$$\text{figuraFinal} = \begin{bmatrix} \text{fila } 1 \\ \vdots \\ \text{fila } m' \end{bmatrix}$$

Los índices m' y n' denotan las nuevas dimensiones de la matriz después de la evolución.

8.2. Ejemplos de máquinas de Turing

8.2.1. Duplicador de unos

Queremos implementar una máquina de Turing cuya funcionalidad sea la de duplicar los unos que aparecen en una cadena de símbolos inicial. El lenguaje utilizado en este ejemplo está formado por '0', '1' y '2'. Para ello utilizamos el autómata representado en la figura 2.1. La forma de representar las cintas en nuestra máquina construida con los patrones del Juego de la Vida son dos pilas que, como hemos visto con más detalle en secciones anteriores, simulan el movimiento de la cinta a través de apilar y desapilar los símbolos. Entonces, podemos representar la cinta como podemos ver en la figura 8.1, donde ambas columnas representan las dos pilas y el rectangulo rojo el símbolo donde está el cabezal de lectura y escritura.

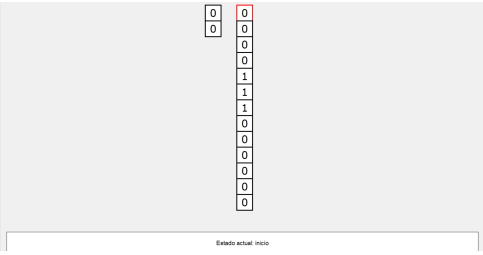

Figura 8.1: Configuración inicial de las pilas para la máquina de Turing que duplica los 1 en una cadena

Así, la cinta sigue en estado de inicio hasta llegar al primer '1'. Después de leer el primer '1', pasa al estado S0, donde como vimos anteriormente en la tabla de transición, la máquina de estados finitos produce un '2' que se apila en la pila derecha y nos movemos a la izquierda en la cinta, lo que es equivalente a desapilar en la pila izquierda y apilar en la derecha el mismo

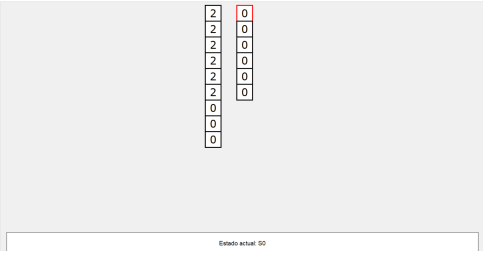

Figura 8.2: Configuración de las pilas para la máquina de Turing que duplica los 1 en una cadena tras transformar cada 1 en dos 2

símbolo. En este caso, el estado que leemos es '0' y, por lo tanto, pasamos al estado S1. Después, escribe otro símbolo '2' y se mueve a la derecha en la cinta. Moverse a la derecha en la cinta es equivalente a desapiliar en la derecha y apilar en la izquierda el mismo símbolo. Así por cada '1' que se encuentra escribe dos símbolos '2' como se puede ver en la figura 8.2 Y posteriormente, por cada símbolo '2' escribe un símbolo '1', consiguiendo duplicar así el número de '1' que había en la cinta original, ya que en el estado S2 lo que indica la tabla de transición es: si se lee un '2' escribimos un '1' y nos movemos a la izquierda. En cambio, si nos encontramos un '0' es porque hemos terminado de duplicar los '1' y termina el programa, ya que llegamos a un estado final.

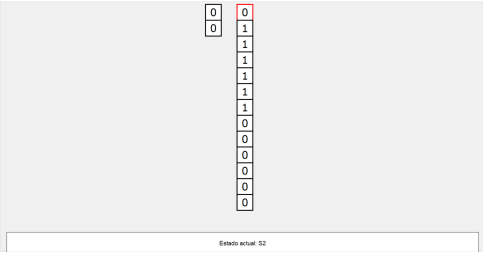

Figura 8.3: Configuración final de las pilas para la máquina de Turing que duplica los 1

8.2.2. Construcción del número un tercio

Queremos implementar una máquina de Turing cuya funcionalidad sea simular el número $\frac{1}{3}$ representado en binario, esto es una cadena infinita de '0' y '1'. En este caso, la salida de la cinta es independiente de la cadena de entrada y, además, no terminará nunca, ya que este número tiene infinitos decimales. Para representar su funcionamiento se utiliza el autómata que muestra la figura 8.4.

La máquina comienza en el estado q0 y pasa al estado q1 escribiendo un '0' y moviendo el cabezal a la derecha sin importar qué símbolo lee. Después, se mueve a la derecha y pasa a q2, escribe un '1' y se mueve a la derecha. En q3 se mueve a la derecha y vuelve al principio. Creando así un ciclo infinito pasando por los cuatro estados.

8.2.3. Construcción de una cadena infinita de unos

Queremos implementar una máquina de Turing cuya funcionalidad sea crear una sucesión de '0' y '1' donde cada vez añadimos un '1' . La diferencia con la máquina anterior es que depende de lo que hayamos escrito previamente. Por ello, es necesario moverse a izquierda y derecha y utilizar otros símbolos, que en este caso son '@', ' ' y 'x'. Podemos representar el comportamiento de esta máquina con el autómata de la figura 8.6.

Y el comportamiento es, de manera intuitiva, escribir '1' y detrás de ellos una 'x' para que cuando toca escribir el siguiente conjunto de '1' escribir tantos como 'x' haya y uno más. El símbolo '@' se usa para marcar el inicio de la cinta, ya que cuando va a escribir los símbolos

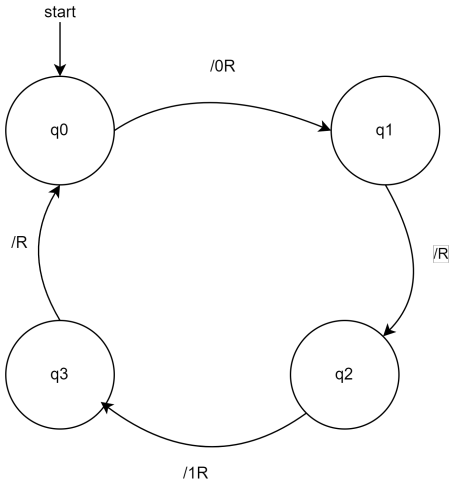

Figura 8.4: Autómata que modela el comportamiento de la máquina que construye el número un tercio

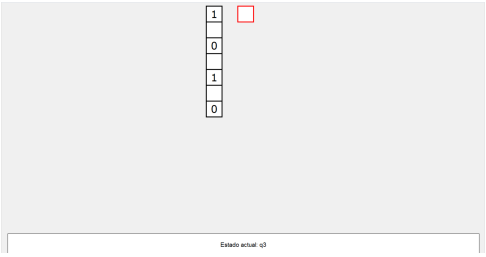

Figura 8.5: cinta de la máquina que construye el número un tercio

'x' antes vuelve hasta el principio y después se mueve hasta la última sucesión de unos. Así, de manera infinita va creando la sucesión que queríamos. De nuevo, la ejecución de esta máquina no depende de la cadena inicial y no tiene estado final.

8.2.4. Suma y producto

Podemos construir también máquinas capaces de sumar y multiplicar números representados en binario. Aunque no se va a detallar su comportamiento, podemos ver un ejemplo de cada una. Para el caso de la suma, realizamos la suma de los números 6 y 5, que en binario son 110 y 101 respectivamente. Es decir, el texto que le pasamos como entrada será '110+101=' y obtenemos la salida que podemos ver en 8.8 que efectivamente es 1011, que es la representación del número 11 en binario.

En el caso del producto, realizamos la multiplicación 6*5. Como cadena inicial tenemos: '110*101='. Como resultado obtenemos lo que se puede ver en 8.9 que, efectivamente, es 11110, que es la representación del número 30 en binario.

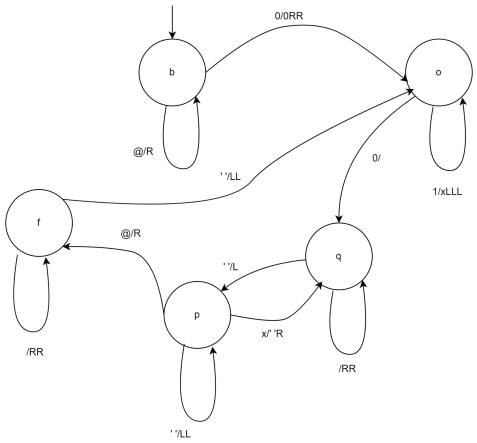

Figura 8.6: Autómata que modela el comportamiento de la máquina que construye una cadena añadiendo cada veces más 1

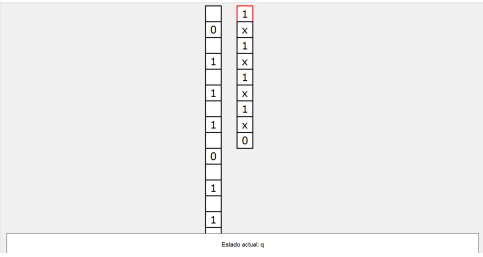

Figura 8.7: Cintas de la máquina que construye una cadena añadiendo cada veces más 1

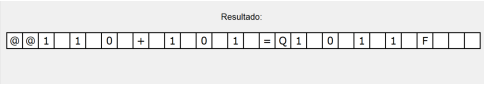

Figura 8.8: Resultado de sumar 5 + 6 en binario

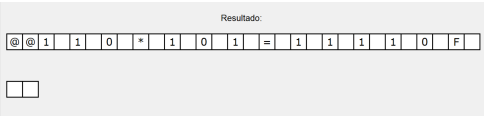

Figura 8.9: Resultado de sumar 5 * 6 en binario

8.3. Código

```
from tkinter import *
import tkinter.font as tkFont
import numpy as np
import time

class Patron():
    def __init__ (self, nombre):
        self.nombre = nombre
        self.ventana = None
        self.figura =  None
    def muestra(self):
        self.ventana = Marco(self.nombre, len(self.figura[0]), len(self.figura) ,
     self)

        for i in range(0,len(self.figura)):
            for j in range(0,len(self.figura[0])):
                if self.figura[i][j]==1:
                    self.ventana.pintaRectangulo (i,j, relleno='red', borde = '
    white')
        self.ventana.mainloop()
    def guarda (self, dirBase,nombrFich):
        fichero = open(dirBase+nombrFich + '.txt','w')
        for linea in self.figura:
            for i in linea:
                fichero.write('%d '%i)
            fichero.write('\n')

class PatronAtomico (Patron):
    def __init__(self, nombrFich, muestra=False):
        super().__init__(nombrFich)
        fichero = open(nombrFich+'.txt')
        self.figura = []
        for l in fichero:
            fila = []
            for i in l.split():
                fila.append(int(i))
            self.figura.append(fila)
        if (muestra):
            self.muestra()

class PatronCompuesto (Patron):
    def __init__(self, nombre, listaPiezas, muestra=False):
        print (nombre, type(listaPiezas), type(listaPiezas[0]),muestra)
        super().__init__(nombre)
        print (listaPiezas)

        nFilas = max ([len(pieza.figura) for pieza in listaPiezas])
        nCol =  max ([len(pieza.figura[0]) for pieza in listaPiezas])
        print ("'"+nombre+"'", nFilas, nCol)
        self.figura=[]
        for y in range(0,nFilas):
            self.figura= self.figura + [[0]*nCol]
        for pieza in listaPiezas:
            for nF in range(0, len(pieza.figura)):
                for nC in range(0, len(pieza.figura[0])):
                    self.figura[nF][nC] = self.figura[nF][nC]+pieza.figura[nF][nC]
        if (muestra):
            self.muestra()

    def figura(self):
        return self.figura
```

```
60
61 class Pieza ():
62     def __init__(self, patron):
63         self.patron = patron
64         self.figura = patron.figura.copy()
65
66     def traslacion (self,nFilas, nCol):
67         #print (self.figura,len(self.figura))
68         figura=self.figura.copy()
69         yp, xp= [len(figura),len(figura[0])]
70         Ly, Lx= [len(self.figura)+nFilas, len(self.figura[0])+nCol]
71
72         self.figura = np.zeros((Ly,Lx)).tolist()
73         for y in range(nFilas, Ly):
74             self.figura[y][nCol:Lx] = figura[y - nFilas]
75
76         return self
77
78     def reflexionEjeX (self):
79         if sum (sum(x) for x in self.figura )==0:
80             self.figura = self.patron.figura.copy()
81         nF = int(len(self.figura))
82         for fila in range(0,int (nF/2)):
83             self.figura[fila], self.figura[nF-fila -1] = (self.figura[nF-fila -1].
    copy(), self.figura[fila].copy())
84         return self
85
86     def reflexionEjeY (self):
87         if sum (sum(x) for x in self.figura )==0:
88             self.figura = self.patron.figura.copy()
89         nF = int(len(self.figura))
90         for fila in range(0,nF):
91             self.figura[fila]= self.figura[fila].copy()[::-1]
92         return self
93
94     def rotacion (self, n):
95         for i in range(0,n):
96             npfig = np.array(self.figura.copy())
97             npfigOut = npfig.transpose()
98             for fila in range(0,npfig.shape[0]):
99                 npfigOut [:,fila] = npfig[fila,:][::-1]
100            self.figura = npfigOut.tolist()
101        return self
102
103    def __cuentaVecinos__(self, i, j):
104        nFilas = len(self.figura)
105        nCol = len(self.figura[0])
106        """for iF in range(max(0, i - 1), min(self.nFilas, i + 2)):
107            for iC in range(max(0, j - 1), min(self.nCol, j + 2)):
108                print(self.configuracion[iF][iC], end='\t')
109            print()
110        print()"""
111        suma = 0
112        for iF in range(max(0, i - 1), min(nFilas, i + 2)):
113            for iC in range(max(0, j - 1), min(nCol, j + 2)):
114                suma = suma + self.figura[iF][iC]
115        suma = suma - self.figura[i][j]
116        return suma
117
118    def __siguiente__(self):
119        nFilas = len(self.figura)
120        nCol = len(self.figura[0])
121        config = []
```

```
122         for linea in self.figura:
123             config.append(linea[:])
124 
125         cambios = 0
126         vivos =[]
127         muertes =[]
128         for i in range (0, nFilas):
129             for j in range (0, nCol):
130                 nVecinos = self.__cuentaVecinos__ (i,j)
131                 #print ('nVecinos ',nVecinos)
132                 #if self.configuracion[i][j] ==1: print (i,j, ':', self.
    configuracion[i][j], nVecinos)
133                 if config[i][j]== 0:
134                     if nVecinos==3:
135                         #print (i,j, 'nace')
136                         vivos.append((i,j))
137                         config[i][j] = 1
138                         cambios = cambios +1
139                 else:
140                     if  nVecinos >3 or nVecinos < 2:
141                         #print(i, j, 'muere')
142                         muertes.append((i, j))
143                         config[i][j] = 0
144                         cambios = cambios + 1
145                     else:
146                         vivos.append((i,j))
147         #print("\t\tPinto", config[4:7])
148         self.figura = config.copy()
149 
150         return cambios
151 
152     def evolve (self, nSteps):
153 
154         for i in range(0, nSteps):
155 
156             Lx = len(self.figura[0])
157             self.figura = [[0] * Lx] + self.figura + [[0] * Lx]
158 
159             for fila in range(0, len(self.figura)):
160                 self.figura[fila] = [0]  + self.figura[fila] + [0]
161 
162             self.__siguiente__()
163 
164             npfig = np.array(self.figura)
165             while sum(npfig[:,0])==0:
166                 npfig = npfig[:,1:].copy()
167             while sum(npfig[:,-1])==0:
168                 npfig = npfig[:,:-1].copy()
169             while sum(npfig[0,:])==0:
170                 npfig = npfig[1:,:].copy()
171             while sum(npfig[-1,:])==0:
172                 npfig = npfig[:-1,:].copy()
173             self.figura = npfig.tolist()
174         return self
175 
176 class Marco(Frame):
177     def __init__(self, nombre, ancho, alto, objeto=None):
178         print('ancho',ancho, 'alto',alto)
179         alturaBotones = 40
180         root = Tk()
181         self.objeto = objeto
182 
183         self.master = root
```

```
184         self.master.title('Game of Life: ' + nombre)
185         self.master.resizable(width=False, height=False)
186         escala = min(root.winfo_screenwidth() * 0.9 / ancho, (root.
    winfo_screenheight()-alturaBotones) * 0.9 / alto)
187         escala = int(escala)
188         dim = escala * ancho, escala * alto + alturaBotones
189         print('Dimensiones:', dim)
190         root.geometry(str(dim[0]) + 'x' + str(dim[1]))
191 
192         self.dim = dict(x0=10, y0=10, deltaX=50, deltaY=50)
193         self.fuente = [tkFont.Font(family='Verdana', size=int(self.dim['deltaX'] /
     2), weight='normal'),
194                        tkFont.Font(family='Verdana', size=int(self.dim['deltaX'] /
     2), weight='bold')]
195         self.color = ['black', 'red']
196         self.objetoMostrado = [1]
197         self.longitud = escala
198 
199         Frame.__init__(self, root)
200         self.grid()
201         # self.creaWidgets()
202         self.canvas1 = Canvas(self.master, width=self.master.winfo_screenwidth(),
203                               bg="#FFFFFF", height=self.master.winfo_screenheight
    () )
204         self.canvas1.grid(sticky='ew')
205         print('Canvas', self.canvas1.winfo_width(), self.canvas1.winfo_height())
206 
207         self.canvas2 = Canvas(self.master, width=self.master.winfo_screenwidth(),
208                               height=alturaBotones)
209         self.canvas2.grid(sticky='nsew')
210 
211         self.master.grid_rowconfigure(1, weight=1)
212         self.master.grid_columnconfigure(0, weight=1)
213 
214 
215     def pintaRectangulo(self, i, j, relleno, borde):
216         #print (i,j,relleno, borde)
217         x = j * self.longitud
218         y = i * self.longitud
219         self.canvas1.create_rectangle(x, y, x + self.longitud, y + self.longitud,
220                                       fill=relleno, outline=borde,tags='1')
221 
222 class GameOfLife():
223     def __init__(self, directorio, fichero, retardo, cuadricula=False, sombrea=
    False, nIter=None):
224         segmento0 = open(directorio + fichero + '.txt')
225         configuracion = []
226         for l in segmento0:
227             fila = []
228             for i in l.split():
229                 #print(i, int(i))
230                 fila.append(int(i))
231             #print(fila)
232             #configuracion.append([0]+fila)
233             configuracion.append( fila)
234         #print (configuracion)
235 
236         self.configuracion = configuracion
237         self.NITERMAX = nIter
238         self.nFilas = len(configuracion)
239         self.nCol = len(configuracion[0])
240         self.muertes = set()
241         self.sombrea = sombrea
```

```
242
243         print('Tamano:',len(configuracion), len(configuracion[0]))
244
245         self.retardo = retardo
246         self.activo = False
247         self.ventana = Ventana03(fichero, len(configuracion[0]), len(configuracion
    ), self, cuadricula)
248         for i in range(0,len(configuracion)):
249             for j in range(0,len(configuracion[0])):
250                 if configuracion[i][j]==1:
251                     self.ventana.pintaRectangulo (i,j, relleno='red', borde = '
    white')
252
253         self.ventana.update()
254         #self.ventana.mainloop()
255         self.evolve()
256     def __cuentaVecinos__(self, i, j):
257         """for iF in range(max(0, i - 1), min(self.nFilas, i + 2)):
258             for iC in range(max(0, j - 1), min(self.nCol, j + 2)):
259                 print(self.configuracion[iF][iC], end='\t')
260             print()
261         print()"""
262         suma = 0
263         for iF in range(max(0, i - 1), min(self.nFilas, i + 2)):
264             for iC in range(max(0, j - 1), min(self.nCol, j + 2)):
265                 suma = suma + self.configuracion[iF][iC]
266         suma = suma - self.configuracion[i][j]
267         return suma
268
269     def __siguiente__(self):
270         config = []
271         for linea in self.configuracion:
272             config.append(linea[:])
273
274         cambios = 0
275         nacimientos =[]
276         #muertes =[]
277         for i in range (0, self.nFilas):
278             for j in range (0, self.nCol):
279                 nVecinos = self.__cuentaVecinos__ (i,j)
280                 #if self.configuracion[i][j] ==1: print (i,j, ':', self.
    configuracion[i][j], nVecinos)
281                 if self.configuracion[i][j]== 0:
282                     if nVecinos==3:
283                         #print (i,j, 'nace')
284                         nacimientos.append((i,j))
285                         config[i][j] = 1
286                         cambios = cambios +1
287                 else:
288                     if  nVecinos >3 or nVecinos < 2:
289                         #print(i, j, 'muere')
290                         self.muertes.add((i, j))
291                         config[i][j] = 0
292                         cambios = cambios + 1
293                     else:
294                         nacimientos.append((i,j))
295         #print("\t\tPinto", config[4:7])
296         self.configuracion = config
297         #print ('Nacen:', len(vivos), ' Mueren:', len(muertes))
298         self.ventana.canvas1.delete('rect')
299         """for i,j in muertes:
300             self.ventana.pintaRectangulo (i,j, relleno='white', borde = 'white')
    """
```

```
301         if self.sombrea:
302             for i,j in self.muertes:
303                 self.ventana.pintaRectangulo (i,j, relleno='#999999', borde = '
    white')
304         for i, j in nacimientos:
305             self.ventana.pintaRectangulo(i, j, relleno='red', borde='white')
306         return cambios
307 
308     def evolve (self):
309         #print ("Pinto", self.configuracion[4:7])
310         #self.ventana.pintaMatriz(self.configuracion)
311         #cambios = self.__siguiente__()
312         #self.ventana.pintaMatriz(self.configuracion)
313         #print("\tPinto", self.configuracion[4:7])
314         #print("cambios=",cambios)
315         nIter = 0
316         cambios = 1
317         #i=0
318         while cambios > 0 and (self.NITERMAX == None or (self.NITERMAX != None and
     nIter < self.NITERMAX)):
319         #for i in range(0,10):
320             if self.activo:
321                 cambios = self.__siguiente__()
322                 #print ('i=',i)
323                 #i=i+1
324                 #self.ventana.pintaMatriz(self.configuracion)
325                 #print (nIter)
326                 self.ventana.texto.delete('1.0', 'end')
327                 self.ventana.texto.insert('1.0', str(nIter),'center')
328                 time.sleep(self.retardo)
329                 #self.ventana.texto.delete('1.0','end')
330                 nIter = nIter + 1
331 
332 
333             self.ventana.update()
334 
335 
336         self.ventana.mainloop()
337 
338 class Ventana03(Frame):
339     def __init__(self, nombre, ancho, alto, objeto=None, cuadricula=True ):
340         print('ancho',ancho, 'alto',alto)
341         alturaBotones = 40
342         root = Tk()
343         self.objeto = objeto
344 
345         self.master = root
346         self.master.title('Game of Life: ' + nombre)
347         self.master.resizable(width=False, height=False)
348         escala = min(root.winfo_screenwidth() * 0.9 / ancho, (root.
    winfo_screenheight()-alturaBotones) * 0.9 / alto)
349         escala = int(escala)
350         dim = escala * ancho, escala * alto + alturaBotones
351         print('Dimensiones:', dim)
352         root.geometry(str(dim[0]) + 'x' + str(dim[1]))
353 
354         self.dim = dict(x0=10, y0=10, deltaX=50, deltaY=50)
355         self.fuente = [tkFont.Font(family='Verdana', size=int(self.dim['deltaX'] /
     2), weight='normal'),
356                        tkFont.Font(family='Verdana', size=int(self.dim['deltaX'] /
     2), weight='bold')]
357         self.color = ['black', 'red']
358         self.objetoMostrado = [1]
```

```
359         self.longitud = escala
360 
361         Frame.__init__(self, root)
362         self.grid()
363         # self.creaWidgets()
364         self.canvas1 = Canvas(self.master, width=self.master.winfo_screenwidth(),
365                               bg="#FFFFFF", height=self.master.winfo_screenheight
    () )
366         self.canvas1.grid(sticky='ew')
367         print('Canvas', self.canvas1.winfo_width(), self.canvas1.winfo_height())
368 
369         self.canvas2 = Canvas(self.master, width=self.master.winfo_screenwidth(),
370                               height=alturaBotones)
371         self.canvas2.grid(sticky='nsew')
372 
373         self.master.grid_rowconfigure(1, weight=1)
374         self.master.grid_columnconfigure(0, weight=1)
375 
376         canvasBotones = Canvas(self.canvas2, width=self.master.winfo_screenwidth()
    ,
377                                height=30)
378         print ('Activo', self.objeto.activo)
379         #botonOn = Button(canvasBotones, command= (lambda : (self.activa = True) )
     , text="Empezar",
380         self.botonOn = Button(canvasBotones, command=self.buttonOnClick, text="
    Empezar",
381                               width=20, font='Verdana', bg="#DDDDDD")
382         self.botonOff = Button(canvasBotones, command=self.buttonOffClick, text="
    Parar",
383                                width=20, font='Verdana', bg="#DDDDDD", state='disabled'
    )
384         self.texto = Text (canvasBotones,height=canvasBotones.winfo_height(),
385                               width=10, font='Verdana', bg="#DDDDDD")
386         canvasBotones.pack(side='bottom')
387         self.botonOn.pack(side=LEFT, padx=20, pady=5)
388         self.botonOff.pack(side=RIGHT)
389         self.texto.pack (side=RIGHT)
390 
391         if cuadricula:
392             for i in range(0,ancho):
393                 self.canvas1.create_line(    i*self.longitud,0,i*self.longitud,
394                                              root.winfo_screenheight()-
    alturaBotones,
395                                              fill='#0000AA',width=1)
396             for i in range(0,alto):
397                 self.canvas1.create_line(    0, i*self.longitud,
398                                              root.winfo_screenwidth(),i*self.
    longitud,
399                                              fill='#0000AA',width=1)
400 
401     def buttonOnClick(self):
402         """ handle button click event and output text from entry area"""
403         self.botonOn['state'] ='disabled'
404         self.botonOff['state'] ='active'
405         self.objeto.activo = True
406     def buttonOffClick(self):
407         """ handle button click event and output text from entry area"""
408         self.botonOff['state'] ='disabled'
409         self.botonOn['state'] ='active'
410         self.objeto.activo = False
411 
412 
```

```
413     def creaRectangulo(self, canvas, x0, y0, xF, yF, texto, color='black', anchura
    =3, fuente=0):
414         canvas.create_rectangle(x0, y0, xF, yF, fill='white', width=anchura,
    outline=color, tags='1')
415         canvas.create_text((x0 + xF) / 2, (y0 + yF) / 2, text=texto,
416                            font=self.fuente[fuente], fill=self.color[fuente])
417 
418     def muestraLista(self, lista):
419         x0, y0, deltaX, deltaY = self.dim['x0'], self.dim['y0'], self.dim['deltaX'
    ], self.dim['deltaY']
420         nCol = int((self.master.winfo_width() - 2 * self.dim['x0']) / self.dim['
    deltaX'])
421 
422         # print (self.master.winfo_width(), 'nCol=',nCol)
423         print('lista=', lista, nCol)
424         if nCol > 1:
425             for i in range(len(lista)):
426                 print(i)
427                 fila = i // nCol
428                 columna = i % nCol
429                 x = x0 + columna * deltaX
430                 y = self.dim['y0'] + fila * (self.dim['y0'] + self.dim['deltaY'])
431                 self.creaRectangulo(self.canvas1, x, y, x + deltaX, y + deltaY,
    lista[i])
432         print(lista)
433 
434     def pintaMatriz(self, L):
435         canvas = self.canvas1
436         # print ("Pinto matriz", len(L),len(L[0]),canvas, self.canvas1,self.
    longitud)
437         for i in range(0, len(L)):
438             for j in range(0, len(L[0])):
439                 if L[i][j] == 0:
440                     x = j * self.longitud
441                     y = i * self.longitud
442                     self.canvas1.create_rectangle(x, y, x + self.longitud, y +
    self.longitud,
443                                                   fill='white', outline='white')
444                 if L[i][j] != 0:
445                     x = j * self.longitud
446                     y = i * self.longitud
447                     canvas.create_rectangle(x, y, x + self.longitud, y + self.
    longitud,
448                                             fill='red', outline='red')
449 
450     def pintaRectangulo(self, i, j, relleno, borde):
451         #print (i,j,relleno, borde)
452         x = j * self.longitud
453         y = i * self.longitud
454         self.canvas1.create_rectangle(x, y, x + self.longitud, y + self.longitud,
455                                       fill=relleno, outline=borde, tag='rect')
456 
457     def destaca(self, L, i):
458         x0, y0, deltaX, deltaY = self.dim['x0'], self.dim['y0'], self.dim['deltaX'
    ], self.dim['deltaY']
459         print(self.master.winfo_width(), self.master.winfo_height())
460         nCol = int((self.master.winfo_width() - 2 * self.dim['x0']) / self.dim['
    deltaX'])
461         print(self.master.winfo_width(), self.dim['x0'], self.dim['deltaX'], i, '
    nCol=', nCol)
462         if nCol > 1:
463             print('\t', i // nCol)
464             fila = i // nCol
```

```
            columna = i % nCol
            x = x0 + columna * deltaX
            y = self.dim['y0'] + fila * (self.dim['y0'] + self.dim['deltaY'])
            self.creaRectangulo(canvas1, x, y, x + deltaX, y + deltaY,
                                L[i], color='red', anchura=3, fuente=1)

    def casillaNormal(self, L, i):
        x = self.dim['x0'] + i * self.dim['deltaX']
        self.creaRectangulo(x, self.dim['y0'], x + self.dim['deltaX'], self.dim['y0'] + self.dim['deltaY'],
                            L[i], color='black', anchura=3, fuente=0)
```

8.3.1. Código para generar patrones atómicos

```
"""
#%%%%%%%%%%%%%%%%%%%%%%%%
#     Patrones atomicos:
#%%%%%%%%%%%%%%%%%%%%%%%%
"""
pBloque = PatronAtomico (dirBase + 'patronesAtomicos/'+'bloque')
pSegmento = PatronAtomico (dirBase + 'patronesAtomicos/'+'segmento')
pAnnihilator = PatronAtomico (dirBase +'patronesAtomicos/'+'aniquilador01')
pBloqueAmpliado = PatronAtomico (dirBase +'patronesAtomicos/'+'bloqueAmpliado')
pPreGlider = PatronAtomico(dirBase+'patronesAtomicos/'+'debris')
pNave = PatronAtomico(dirBase+'patronesAtomicos/'+'nave')
pPatronesAtomicos = PatronCompuesto (
    nombre='Patrones atomicos',
    listaPiezas=[   Pieza (pBloque).traslacion(1,1),
                    Pieza (pAnnihilator).traslacion(1,4),
                    Pieza (pSegmento).traslacion(0,7),
                    Pieza (pBloqueAmpliado).traslacion(1,13),
                    Pieza (pPreGlider).evolve(1).reflexionEjeY().traslacion(1,17),
                    Pieza (pNave).traslacion(1,21)
                ],
    muestra = False
)
pPatronesAtomicos.guarda(dirPEjempl,'patronesAtomicos')
```

8.3.2. Código para generar patrones que se desplazan

```
"""
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#     Patrones que se desplazan:
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
"""
pPreGlider = PatronAtomico(dirBase+'patronesAtomicos/'+'debris')
pNave = PatronAtomico(dirBase+'patronesAtomicos/'+'nave')
pMoviles = PatronCompuesto(nombre='patrones moviles',
            listaPiezas=[
                Pieza(pNave).traslacion(1, 1),
                Pieza(pPreGlider).reflexionEjeY().evolve(1).traslacion(8, 1)],
            muestra=False)
for i in range(0, len(pMoviles.figura)):
    pMoviles.figura[i] = pMoviles.figura[i] + [0] * 8

pMoviles.figura = pMoviles.figura + [[0] * len(pMoviles.figura[0])] * 4
pMoviles.guarda(dirBase + 'patronesEjemplo/', 'pMoviles')
#GameOfLife(dirBase + 'patronesEjemplo/', 'pMoviles', 0.1,nIter=12,cuadricula=True
    ,muestraInit=True)

```

```
20
21 pEvol = PatronCompuesto (nombre='Patrones atomicos',
22         listaPiezas=[  Pieza(pAnnihilator).traslacion(1,1)],
23         muestra = False
24         )
25 for i in range(0,len(pEvol.figura)):
26     pEvol.figura[i]= [0]*20 + pEvol.figura[i]+[0]*20
27
28 pEvol.figura = [[0]*len(pEvol.figura[0])]*20 +  pEvol.figura + [[0]*len(pEvol.
    figura[0])]*20
29 pEvol.guarda(dirPEjempl, 'pEvol')
30 #GameOfLife(dirPEjempl, 'pEvol',0.1,nIter=8)
```

8.3.3. Código para generar una pistola y un vaquero

```
1  """
2  # %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
3  #      Vaqueros y pistola
4  # %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
5  """
6
7  pQueenBee = PatronCompuesto (nombre ='Queen Bee',
8              listaPiezas =  [
9                      Pieza(pBloque),
10                     Pieza (pBloqueAmpliado).traslacion(2,3),
11                     Pieza (pBloque).traslacion(5,0)],
12             muestra=False
13                        )
14
15 p1 = PatronCompuesto(nombre='bee + block',
16     listaPiezas=[
17         Pieza(pBloque).traslacion(2, 0),
18         Pieza(pQueenBee).evolve(10).reflexionEjeY().traslacion(0, 4)],
19     muestra=False)
20
21 pPistola = PatronCompuesto (  nombre='pistola',
22             listaPiezas=[
23                 Pieza(p1).evolve(10).reflexionEjeY().traslacion(0,20),
24                 Pieza(p1).evolve(15).traslacion(2,0)],
25             muestra=False)
26
27
28 pAbejaConfinada01 = PatronCompuesto (  nombre='abeja confinada #01',
29                     listaPieza=[
30                         Pieza(pAnnihilator).reflexionEjeX().reflexionEjeY().
    traslacion(7,0),
31                         Pieza(pQueenBee).evolve(10).traslacion(2,11),
32                         Pieza(pAnnihilator).traslacion(0,20)],
33                         muestra=False)
34
35 pAbejaConfinada02 = PatronCompuesto (  nombre='abeja confinada #02',
36                     listaPieza=[
37                         Pieza(pAnnihilator).reflexionEjeX().reflexionEjeY().
    traslacion(7,0),
38                         Pieza(pQueenBee).evolve(10).reflexionEjeY().traslacion(2,
    5),
39                         Pieza(pAnnihilator).traslacion(0,19)],
40                     muestra=False)
41
42 pAbejaConfinada03 = PatronCompuesto (  nombre='abeja confinada #03',
43                     listaPiezas=[
```

```
                        Pieza(pAnnihilator).reflexionEjeX().reflexionEjeY().
    traslacion(7,0),
                        Pieza(pQueenBee).evolve(10).reflexionEjeY().reflexionEjeY
    ().traslacion(2,11),
                        Pieza(pAnnihilator).reflexionEjeX().traslacion(7,20)],
                    muestra=False)

pVaqueros = PatronCompuesto (  nombre='Vaqueros',
            listaPiezas=[
                Pieza(pAbejaConfinada01).traslacion(1, 1),
                #Pieza(pAbejaConfinada02).traslacion(15, 1),
                #Pieza(pAbejaConfinada03).traslacion(30, 1)
                ],
            muestra=False)
pVaqueros.guarda(dirPEjempl, 'pVaqueros')
#GameOfLife(dirPEjempl, 'pVaqueros',0.5,nIter=30)

pmuestraPistola = PatronCompuesto (  nombre='Pistola',
                    listaPiezas=[
                        #Pieza(pPistola).traslacion(1, 1),
                        Pieza(pPistola).evolve(60).traslacion(1, 1)
                        ],
                    muestra=False)
```

8.3.4. Código para generar colisiones

```
"""
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#      Colisiones
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
"""
pKickBack = PatronAtomico (dirBase + 'patronesEjemplo/'+'kickback')
pColisiones = PatronCompuesto (  nombre='Colisiones',
                listaPiezas=[
                    Pieza(pKickBack).evolve(67).traslacion(5,3),
                    Pieza(pAbejaConfinada02).evolve(0).traslacion(24,24),
                    #Pieza(pAbejaConfinada03).traslacion(30, 1)
                    ],
                muestra=False)
pColisiones.guarda(dirPEjempl, 'pColisiones')
#GameOfLife (dirBase + 'patronesEjemplo/' ,   'pColisiones', 0.05,nIter=150,
    cuadricula=False)
pGlider = PatronCompuesto (  nombre='Glider',
        listaPiezas=[
            Pieza(pPreGlider).evolve(5).reflexionEjeY(),
            ],
        muestra=False)
palabra ='11111111'
listaPiezas = []
for iGlider in range(0,len(palabra),2):
    desplazamiento = 15 * iGlider//2
    if palabra[iGlider]=='1':
        listaPiezas.append(Pieza(pGlider).traslacion(desplazamiento,desplazamiento
    ))


for iGlider in range(1,len(palabra),2):
    desplazamiento = int ( 30 * iGlider/4)
    if palabra[iGlider]=='1':
        listaPiezas.append(Pieza(pGlider).evolve(2).traslacion(desplazamiento+1,
    desplazamiento))

```

```
34
35 pFlujo = PatronCompuesto(nombre='Flujo',
36                          listaPiezas=listaPiezas,
37                          muestra=False)
38 pFlujo.guarda(dirPEjempl,'pFlujo')
39 pColisiones2 = PatronCompuesto (  nombre='Colisiones',
40             listaPiezas=[
41                 Pieza(pGlider).evolve(2).traslacion(18,5),
42                 Pieza(pGlider).evolve(2).reflexionEjeY().traslacion(19,11),
43                 Pieza(pFlujo).traslacion(0,43),
44                 Pieza(pGlider).evolve(2).reflexionEjeY().traslacion(54,101)
45                 ],
46             muestra=False)
47 pColisiones2.guarda(dirPEjempl, 'pColisiones2')
48 #GameOfLife (dirBase + 'patronesEjemplo/' ,    'pColisiones2', 0.5,nIter=150,
    cuadricula= False)
49
50 ######################################################
51 #   Doble aniquilacion
52 ######################################################
53
54 def creaPalabra(codigo):
55     listaGliders = []
56     saltoVertical = 0
57     for i in range(0,len(codigo)):
58         salto = 7 + (i + 1) % 2
59         for glider in listaGliders:
60             glider=glider.traslacion(0,salto)
61         listaGliders.append(Pieza(pPreGlider).evolve(1+2*i).traslacion(
    saltoVertical,0))
62         saltoVertical = saltoVertical + salto
63     for i in range(len(codigo)-1,-1,-1):
64         print (i, codigo[i])
65         if codigo[i]=='0':
66             npFig = np.array (listaGliders[i].figura)
67             npFig = npFig * 0
68             listaGliders[i].figura = npFig.tolist()
69     return listaGliders
70
71 pBloque = PatronAtomico (dirBase + 'patronesAtomicos/'+'bloque')
72 pBloqueAmpliado = PatronAtomico (dirBase +'patronesAtomicos/'+'bloqueAmpliado')
73
74 pQueenBee = PatronCompuesto (nombre ='Queen Bee',
75           listaPiezas =  [  Pieza(pBloque),
76                             Pieza (pBloqueAmpliado).traslacion(2,3),
77                             Pieza (pBloque).traslacion(5,0)],
78           muestra=False
79             )
80 p1 = PatronCompuesto(nombre='bee + block',
81      listaPiezas=[Pieza(pBloque).traslacion(2, 0),
82                   Pieza(pQueenBee).evolve(10).reflexionEjeY().traslacion(0, 4)],
83      muestra=False)
84
85 pPistola = PatronCompuesto (  nombre='pistola',
86             listaPiezas=[Pieza(p1).evolve(10).reflexionEjeY().traslacion(0,20),
87                          Pieza(p1).evolve(15).traslacion(2,0)],
88             muestra=False)
89
90 pAnnihilator = PatronAtomico (dirBase +'patronesAtomicos/'+'aniquilador01')
91 pPreGlider = PatronAtomico(dirBase+'patronesAtomicos/'+'debris')
92
93
```

```
p1 = PatronCompuesto (nombre='', listaPiezas=[Pieza(PatronCompuesto('',creaPalabra
    ('11001')))],muestra=False)
p11 = PatronCompuesto (nombre='', listaPiezas=[Pieza(PatronCompuesto('',
    creaPalabra('10101')))],muestra=False)
p2= PatronCompuesto(nombre='', listaPiezas=[Pieza(pPistola).evolve(120)],muestra=
    False)
p21= PatronCompuesto(nombre='', listaPiezas=[Pieza(p2).evolve(150)],muestra=False)
prueba = PatronCompuesto( nombre='pistola + aniquilador',
      listaPiezas = [   Pieza(pPistola).evolve(240),
            Pieza(pPreGlider).evolve(4).traslacion(59,82),
            Pieza(pAnnihilator).reflexionEjeX().traslacion(63,85),
            Pieza(pAnnihilator).reflexionEjeX().traslacion(74,88),
            Pieza(pPistola).evolve(150).traslacion(30,0),
            Pieza(pAnnihilator).reflexionEjeX().traslacion(74, 58),
            Pieza(pPreGlider).evolve(1).traslacion(53,44)
                  ],
      muestra=False)
prueba.figura = prueba.figura + [[0]*len(prueba.figura[0])]*2
prueba.guarda (dirPEjempl, 'dobleAniquilacion')
GameOfLife (dirPEjempl, 'dobleAniquilacion',0.1, cuadricula=False)
```

8.3.5. Código para crear una puerta NOT

```
"""
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   Ejemplo NOT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
"""
pNot1 = PatronCompuesto( nombre='pistolaNOTInversa',
          listaPiezas = [Pieza(pPistola).reflexionEjeY().evolve(240)],
          muestra=False)

prueba = PatronCompuesto( nombre='pistolaNOT',
          listaPiezas = [Pieza(pNot1).traslacion(1,len(pNot1.figura[0])+3),
                         Pieza(pPistola).evolve(240),
                         #Pieza(pAnnihilator).rotacion(2).traslacion(75,67)],
                         Pieza(pAnnihilator).rotacion(2).traslacion(85,57)],
          muestra=False)
prueba.guarda(dirBase,'NOT1')

def creaPalabra(codigo):
    listaGliders = []
    saltoVertical = 0
    for i in range(0,len(codigo)):
        salto = 7 + (i + 1) % 2
        for glider in listaGliders:
            glider=glider.traslacion(0,salto)
        listaGliders.append(Pieza(pPreGlider).evolve(1+2*i).traslacion(
    saltoVertical,0))
        saltoVertical = saltoVertical + salto
    for i in range(len(codigo)-1,-1,-1):
        print (i, codigo[i])
        if codigo[i]=='0':
            npFig = np.array (listaGliders[i].figura)
            npFig = npFig * 0
            listaGliders[i].figura = npFig.tolist()
    return listaGliders

p2= PatronCompuesto(nombre='', listaPiezas=[Pieza(pPistola).evolve(240)],muestra=
    False)
p21= PatronCompuesto(nombre='', listaPiezas=[Pieza(PatronCompuesto('',creaPalabra(
    '10011111')))],muestra=False)
```

```
prueba = PatronCompuesto( nombre='Ejemplo NOT #2',
       listaPiezas = [    Pieza(p2),
             Pieza(p21).traslacion(    len(p2.figura)-len(p21.figura)+1,
                                         len(p2.figura[0])+3
                                     )
                     ],
       muestra=False)
Lx = len(prueba.figura[0])
prueba.figura = prueba.figura+ [[0]*Lx]*30
prueba.guarda (dirPComp, 'puertaNOT')
```

8.3.6. Código para crear una puerta AND

```
"""%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
             Puerta AND
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"""

p1 = PatronCompuesto (nombre='', listaPiezas=[Pieza(PatronCompuesto('',creaPalabra('11001')))],muestra=False)
p11 = PatronCompuesto (nombre='', listaPiezas=[Pieza(PatronCompuesto('', creaPalabra('10101')))],muestra=False)
p2= PatronCompuesto(nombre='', listaPiezas=[Pieza(pPistola).evolve(120)],muestra=False)
p21= PatronCompuesto(nombre='', listaPiezas=[Pieza(p2).evolve(150)],muestra=False)
prueba = PatronCompuesto( nombre='Ejemplo AND',
       listaPiezas = [    Pieza(p21),
             Pieza(p1).traslacion(    len(p2.figura)-len(p1.figura)+8,
                                         len(p2.figura[0])+11
                                     ),
             Pieza(p11).traslacion(len(p2.figura) - len(p11.figura) + 8,
                                  len(p2.figura[0]) + 19
                                  ),
             Pieza(pAnnihilator).reflexionEjeX().traslacion(74,88)
                     ],
       muestra=False)
prueba= PatronCompuesto(nombre='', listaPiezas=[Pieza(prueba).reflexionEjeY()], muestra=False)
prueba.figura = prueba.figura + [[0]*len(prueba.figura[0])]*2
prueba.guarda (dirPComp, 'puertaAND')
GameOfLife (dirPComp, 'puertaAND',0.1, cuadricula=False)
```

8.3.7. Código para incrementar en 1 un registro

```
"""
####################################################################################
   Incrementar un registro en 1 (moviendo bloque en diagonal 3 casillas direccion NW)
####################################################################################
"""
pInc1 = PatronCompuesto (     nombre='celdaMemoria',
         listaPiezas = [ Pieza(pPreGlider).evolve(1).reflexionEjeY().traslacion(0,2),
             Pieza(pPreGlider).evolve(3).reflexionEjeY().traslacion(9,0),
             Pieza (pBloque).traslacion(8,5)],
         muestra=False)
pInc1.figura = [[0]*len(pInc1.figura[0])]*1+pInc1.figura + [[0]*len(pInc1.figura[0])]*1

for i in range(0,len(pInc1.figura)):
    pInc1.figura[i]=pInc1.figura[i]+[0]*2
```

```
15
16
17 pInc1.guarda(dirPComp,'pInc1')
18 GameOfLife (dirPComp, 'pInc1',0.1,cuadricula=True,muestraInit=True)
```

8.3.8. Código para decrementar en 1 un registro

```
1  """
2  ##############################################################################################
3     Decrementar un registro en 1/3 (moviendo bloque en diagonal 1 casillas
       direccion SE)
4    Hace falta combinar tres de estos para decrementar el registro en una unidad
5  ##############################################################################################
6  """
7  pPreDecr01 = PatronCompuesto (     nombre='celdaMemoria',
8              listaPiezas = [ Pieza(pPreGlider).evolve(4).reflexionEjeY(),
9                  Pieza(pPreGlider).evolve(4).reflexionEjeY().traslacion(5,0),
10                 Pieza(pPreGlider).evolve(4).reflexionEjeY().traslacion(10,0), #
       Elimina W
11                 Pieza(pPreGlider).evolve(4).reflexionEjeY().traslacion(1,9),  #
       Elimina N
12                 Pieza(pPreGlider).evolve(4).reflexionEjeY().traslacion(17,15),
13                 Pieza (pBloque).traslacion(17,20)
14                             ],
15             muestra=False)
16
17 pPreDecr02 = PatronCompuesto (     nombre='celdaMemoria',
18             listaPiezas = [ Pieza(pPreGlider).evolve(4).reflexionEjeY(),
19                 Pieza(pPreGlider).evolve(4).reflexionEjeY().traslacion(0,5),
20                 Pieza(pPreGlider).evolve(4).reflexionEjeY().traslacion(0,10),
21                 Pieza(pPreGlider).evolve(4).reflexionEjeY().traslacion(9,1),
22                 Pieza(pPreGlider).evolve(4).reflexionEjeY().traslacion(15,17),
23                 #Pieza (pBloque).traslacion(17,19)
24                             ],
25             muestra=False)
26 pDecr = PatronCompuesto (     nombre='celdaMemoria',
27         listaPiezas = [ Pieza(pPreDecr02),
28                         Pieza(pPreDecr01).traslacion(len(pPreDecr02.figura) + 7,
29                                                      len(pPreDecr02.figura[0])
       -3)
30                         ],
31         muestra=False)
32 #pDecr = pPreDecr01
33 pDecr.figura = [[0]*len(pDecr.figura[0])]*1+pDecr.figura + [[0]*len(pDecr.figura
       [0])]*5
34
35 for i in range(0,len(pDecr.figura)):
36     pDecr.figura[i]=[0]*1+pDecr.figura[i]+[0]*10
37
38
39 pDecr.guarda(dirPComp,'pDecrementa1_3')
40 GameOfLife (dirPComp, 'pDecrementa1_3',0.05,cuadricula=True,muestraInit=True)
```

8.3.9. Código para crear una celda de memoria

```
1  """
2  ##############################################################################################
3     Celda base de la memoria, almacenando un byte
4  ##############################################################################################
5  """
```

```
pBloque = PatronAtomico(dirBase + 'patronesAtomicos/' + 'bloque')
pAnnihilator = PatronAtomico(dirBase + 'patronesAtomicos/' + 'aniquilador01')
pBloqueAmpliado = PatronAtomico(dirBase + 'patronesAtomicos/' + 'bloqueAmpliado')
pPreGlider = PatronAtomico(dirBase+'patronesAtomicos/'+'debris')

pQueenBee = PatronCompuesto (nombre ='Queen Bee',
              listaPiezas =  [  Pieza(pBloque),
                Pieza (pBloqueAmpliado).traslacion(2,3),
                Pieza (pBloque).traslacion(5,0)],
              muestra=False
            )

pGlider = PatronCompuesto ( nombre= 'Glider',
        listaPiezas = [Pieza(pPreGlider).evolve(1).reflexionEjeY()],
        muestra = False
        )
p1 = PatronCompuesto(nombre='bee + block',
     listaPiezas=[Pieza(pBloque).traslacion(2, 0),
            Pieza(pQueenBee).evolve(10).reflexionEjeY().traslacion(0, 4)],
     muestra=False)
pAbejaConfinada01 = PatronCompuesto (  nombre='abeja confinada #01',
                listaPiezas=[Pieza(pAnnihilator).reflexionEjeX().reflexionEjeY().
    traslacion(7,0),
                      Pieza(pQueenBee).evolve(10).traslacion(2,11),
                      Pieza(pAnnihilator).traslacion(0,20)],
                muestra=False)

pAbejaConfinada02 = PatronCompuesto (  nombre='abeja confinada #02',
                    listaPiezas=[Pieza(pAnnihilator).reflexionEjeX().reflexionEjeY
    ().traslacion(7,0),
                        Pieza(pQueenBee).evolve(10).reflexionEjeY().traslacion(2,
    5),
                        Pieza(pAnnihilator).traslacion(0,19)],
                    muestra=False)

pAbejaConfinada03 = PatronCompuesto (  nombre='abeja confinada #03',
                    listaPiezas=                    [Pieza(pAnnihilator).
    reflexionEjeX().reflexionEjeY().traslacion(7,0),
                        Pieza(pQueenBee).evolve(10).reflexionEjeY().reflexionEjeY
    ().traslacion(2,11),
                        Pieza(pAnnihilator).reflexionEjeX().traslacion(7,20)],
                    muestra=False)

pPistola = PatronCompuesto (  nombre='pistola',
            listaPiezas=[Pieza(p1).evolve(10).reflexionEjeY().traslacion(0,20),
                Pieza(p1).evolve(15).traslacion(2,0)],
            muestra=False)




pDispersor = PatronCompuesto (  nombre='Dispersor (fanout)',
            listaPiezas=[
                Pieza(pQueenBee).evolve(1).traslacion(3,25),
                Pieza(pAnnihilator).rotacion(2).traslacion(5,0),
                Pieza(pQueenBee).traslacion(0,9),
                Pieza(pAnnihilator).traslacion(1,34),
                #Pieza(pPreGlider).rotacion(1).traslacion(6,18)
            ],
            muestra=False)

pPistola02 = PatronCompuesto (  nombre='pistola 2.0',
            listaPiezas=[Pieza(pAnnihilator).rotacion(2).traslacion(5,0),
```

```
                Pieza(pQueenBee).evolve(10).traslacion(0,10),
                Pieza(pQueenBee).evolve(5).reflexionEjeY().traslacion(2,20),
                Pieza(pAnnihilator).reflexionEjeX().traslacion(7,32)],
            muestra=False)

def creaCelda (palabraByte):
    listaPiezas = [  Pieza(pPistola).evolve(78),
        Pieza(pPistola02).evolve(18).rotacion(3).traslacion(16, 7),
        Pieza(pAbejaConfinada01).evolve(7).rotacion(3).traslacion(52, 3),
        Pieza(pAbejaConfinada02).evolve(17).traslacion(70, 23),
        Pieza(pAbejaConfinada03).evolve(9).rotacion(1).traslacion(40, 40),
        Pieza(pDispersor).rotacion(3).traslacion(25, 20),
        #
        Pieza(pAnnihilator).reflexionEjeX().traslacion(18, 40),
        Pieza(pAnnihilator).rotacion(3).traslacion(32, 45),
    ]
    if palabraByte[7]=='1':
        listaPiezas.append (Pieza(pGlider).evolve(1).reflexionEjeY().traslacion
    (50, 13))
        listaPiezas.append (Pieza(pGlider).rotacion(1).traslacion(31, 36))
    if palabraByte[6]=='1':
        listaPiezas.append( Pieza(pGlider).evolve(1).reflexionEjeY().rotacion(2).
    traslacion(39, 30))
        listaPiezas.append( Pieza(pPreGlider).traslacion(43,20) )
    if palabraByte[5]=='1':
        listaPiezas.append (Pieza(pGlider).rotacion(2).traslacion(46, 28))
    if palabraByte[4]=='1':
        listaPiezas.append(Pieza(pGlider).rotacion(1).reflexionEjeY().traslacion
    (53, 36))
    if palabraByte[3]=='1':
        listaPiezas.append(Pieza(pGlider).evolve(1).rotacion(1).traslacion(62, 36)
    )
    if palabraByte[2]=='1':
        listaPiezas.append(Pieza(pGlider).evolve(1).reflexionEjeX().traslacion(70,
     29))
    if palabraByte[1]=='1':
        listaPiezas.append(Pieza(pGlider).evolve(2).traslacion(67, 20))
    if palabraByte[0]=='1':
        listaPiezas.append(Pieza(pGlider).traslacion(59, 13))
    pCelda = PatronCompuesto (nombre='Celda',
                              listaPiezas= listaPiezas,
                              muestra=False)
    return pCelda

pByte = '10010011'
pCelda = creaCelda(pByte)
for linea in range(0, len(pCelda.figura)):
    pCelda.figura[linea] = [0] + pCelda.figura[linea] + [0]
    pCelda.figura[linea] = [0] + pCelda.figura[linea] + [0]


pCelda.figura = [[0]*len(pCelda.figura[0])] + pCelda.figura + [[0]*len(pCelda.
    figura[0])]
pCelda.guarda(dirPComp,'pCelda'+pByte)
GameOfLife (dirPComp, 'pCelda'+pByte,0.01,cuadricula=False,sombrea=True,nIter=240)
```

8.3.10. Código para seleccionar una celda

```
"""
##############################################################################
   Animacion: Seleccion de celda de memoria
##############################################################################
```

```
"""
pLWSS = PatronAtomico (dirPAtom+'LWSS')
pMWSS = PatronAtomico (dirPAtom+'MWSS')
pPd = PatronAtomico (dirPAtom+'pentaDecathlon')
pCelda = PatronAtomico (dirPComp+'pCelda'+'11111111')
array=np.array(pCelda.figura)
while sum(array[:,-1])==0:
    array = array[:,:-1]

while sum(array[:,0])==0:
    array = array[:,1:]

while sum(array[-1, :]) == 0:
    array = array[:-1, :]

while sum(array[0, :]) == 0:
    array = array[1:, :]

pCelda.figura = list(array)
print (len(pCelda.figura), len(pCelda.figura[0]))

pSeleccion = PatronCompuesto (  nombre='Seleccion celda memoria',
            listaPiezas=[   Pieza(pMWSS).rotacion(3).evolve(4).traslacion(19,0),
                Pieza(pCelda).traslacion(26,6),
                Pieza(pPd).evolve(4).rotacion(1).traslacion(0,
                                                    18 + len(pCelda.figura[0])),
                Pieza(pLWSS).evolve(1).traslacion(17+len(pCelda.figura),
                                                    26 + len(pCelda.figura[0]))],
            muestra=False)

for linea in range(0, len(pSeleccion.figura)):
    pSeleccion.figura[linea] = [0]*5+pSeleccion.figura[linea] + [0] * 30

pSeleccion.figura = [[0]*len(pSeleccion.figura[0])]*6 + pSeleccion.figura + [[0]*
    len(pSeleccion.figura[0])]*6

pSeleccion.guarda(dirPComp,'pSeleccionAnimacion')

GameOfLife (dirPComp, 'pSeleccionAnimacion',0.001,cuadricula=False,sombrea=True,
    nIter=600)
```