INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

TRABAJO TERMINAL

LOS SISTEMAS DE ETIQUETAS Y SU DINÁMICA ESPACIAL EN AUTÓMATAS CELULARES

PRESENTA:

ALAN BARUCH CERNA GONZÁLEZ

DIRECTORES:

DOCTOR GENARO JUÁREZ MARTÍNEZ

Ciudad de México
2021

Instituto Politécnico Nacional

Escuela Superior de Cómputo
Subdirección Académica

No de TT: 2020 - B100 Ciudad de México, 08 de diciembre de 2021

Documento Técnico

Los Sistemas de Etiquetas y su Dinámica espacial en Autómatas Celulares

Presenta:

Alan Baruch Cerna González[1]

Directores:

Doctor Genaro Juárez Martínez

RESUMEN

Dentro del campo de la teoría de la computación hay cosas aún por descubrir que pueden sustentar cada vez más el conocimiento acerca de la computación. Una de ellas es la máquina de Post canónica normal mejor conocida como sistema de etiquetas (sistema tag) la cual, con un simple proceso de manipulación de caracteres, tiene el mismo potencial que una máquina de Turing. Mediante un análisis de la dinámica espacial en las cadenas de símbolos generadas por los sistemas tag es posible encontrar un autómata celular que simule un sistema tag. El análisis espacial de los sistemas tag ofrece un camino para entender mejor su espacio de soluciones. Este documento se conforma por el estado del arte donde se muestra una recopilación de algunos hallazgos realizados a lo largo del siglo pasado y presente, el marco teórico donde se definen en principio los principales conceptos de esta investigación, los hallazgos sobre sistemas tag, su representación gráfica y espacial así como la demostración del funcionamiento de algunos autómatas celulares, además el análisis, diseño e implementación del simulador junto con pantallas de la interfaz gráfica del mismo, y las exploraciones realizadas en algunos autómatas celulares que simulan sistemas tag.

Palabras clave: Sistema Tag, Autómata Celular, Teoría de la Computación, Sistemas Dinámicos, Matemáticas Discretas

[1]acernag1500@alumno.ipn.mx

Advertencia

"Este documento contiene información desarrollada por la Escuela Superior de Cómputo del Instituto Politécnico Nacional, a partir de datos y documentos con derecho de propiedad y por lo tanto, su uso quedará restringido a las aplicaciones que explícitamente se convengan."

La aplicación no convenida exime a la escuela su responsabilidad técnica y da lugar a las consecuencias legales que para tal efecto se determinen.

Información adicional sobre este reporte técnico podrá obtenerse en: La Subdirección Académica de la Escuela Superior de Cómputo del Instituto Politécnico Nacional, situada en Av. Juan de Dios Bátiz s/n Teléfono: 57296000, extensión 52000.

Índice general

Índice de figuras

Índice de tablas

Introducción

En la teoría de la computación se tienen modelos que han sido fundamentales para la tecnología y avance científico que hay hasta la actualidad, algunos ejemplos son: la máquina de Turing, la máquina de Post y los autómatas celulares.

En 1920 Emil Post había desarrollado un modelo computacional logístico de reescritura de cadenas que es una forma simple de la máquina de Post canónica normal, este modelo consiste en que, dada una cadena, lee el primer símbolo, borra un número constante (definido por un número de borrado P) de símbolos al principio y agrega, según el símbolo encontrado al principio, una cadena de símbolos al final (llamada regla de producción), esto sucede hasta que ya no hay un número suficiente de símbolos a borrar o se llegó a un símbolo de parada. Post no lo había publicado hasta 1943 para resolver una reducción del problema de la parada el cual ya había embozado en su escrito de 1921 que no publicó. Este es el sistema conocido como sistema de etiquetas (sistema tag), es determinista y comprende un modelo universal de computación.

En la segunda mitad del siglo XX, John von Neumann desarrolla los autómatas celulares cuyo funcionamiento se describe como: una rejilla llena de celdas con ciertos valores, cada celda es influenciada por las celdas cercanas y dependiendo del valor en estas, la celda en cuestión cambiará de valor. Von Neumann los ideó como sistemas de dos dimensiones sin explorarlos en una dimensión, algo que posteriormente hizo Stephen Wolfram al crear los autómatas celulares elementales que se basan en

la misma dinámica, pero en un espacio unidimensional.

Los sistemas dinámicos son conjuntos de elementos que se relacionan entre sí y a su vez van cambiando con el tiempo. De esta forma se pueden encontrar muchos sistemas dinámicos en muchas áreas de conocimiento y el estudio teórico de estos ha llevado a resultados muy importantes en la biología, la física, la sociología, la computación, entre otros. El estudio de los sistemas dinámicos está soportado por las matemáticas, hecho por el cual su nivel de abstracción es muy elevado. Así, es posible estudiar cualquier sistema que cambie con el tiempo como muchos modelos de computación, y en el sentido de este trabajo, a los sistemas tag. Es decir, para observar la dinámica espacial de los sistemas tag, se aplicaran métodos y conceptos de sistemas dinámicos.

Los autómatas celulares han sido utilizados para modelado discreto de sistemas dinámicos y complejos, el análisis de estos ha resultado sumamente importante para observar el comportamiento de este tipo de sistemas. Su capacidad de computación paralela y la simplicidad que los caracteriza los convierte en una herramienta funcional y eficiente por lo que su aplicación para este trabajo terminal es conveniente.

Capítulo 1

Estado del Arte

1.1. La Computación Universal y los Sistemas Tag

En teoría de la computación se dice que un sistema de reglas de manipulación de datos es computacionalmente universal o Turing-completo cuando este puede simular una máquina de Turing universal, esto debido a que una máquina de Turing universal es capaz de realizar cualquier cálculo, tal como todos los lenguajes de programación actuales.

Los sistemas tag simulan máquinas de Turing universales, esto fue demostrado por Hao Wang en 1963 [10] y por John Cooke con Marvin Minsky en el mismo año [1], su demostración fue para sistemas tag con número de borrado P = 2. Además, se ha hecho lo inverso para demostrar que una máquina de Turing es universal se ha simulado con un sistema tag universal como lo hizo Yu Rogozhin en 1996 [15].

1.2. Dinámica Espacial de Modelos Computacionales por Wolfram

En el 2002, Stephen Wolfram hace un análisis de la dinámica espacial de varios modelos computacionales poniendo cada transición de la simulación debajo de otra para observar su evolución, en este análisis incluye brevemente a los sistemas tag entre otras máquinas de computación, en cuyo análisis plantea un sistema tag que simula el autómata celular elemental de la regla 30 [11].

Con esta investigación Wolfram abrió el paso al análisis espacial y sistemático de modelos computacionales como se hace usualmente para el estudio de los autómatas celulares, algo que metodológicamente será importante para este Trabajo Terminal.

1.3. Sistemas Tag Cíclicos

Matthew Cook, en 2004, diseñó los sistemas tag cíclicos para su demostración de la universalidad en el autómata celular elemental de la regla 110 con el que simuló un sistema tag cíclico Turing-completo. Probando así que al menos un autómata celular elemental es computacionalmente universal [2].

Para este modelo Cook convirtió el número de borrado P por un número de ciclos P y el alfabeto de símbolos a uno binario (0, 1) para que, por cada transición, el primer carácter de la cadena es 1 añade la cadena al final correspondiente al ciclo en el que esta (regla de producción) y si es 0 no lo hará, borrando solo un carácter por ciclo. Este modelo es equivalente al clásico.

1.4. Sistemas Tag en la Computación

Algunas investigaciones del siglo actual han incrementado lo que se sabe sobre los sistemas tag como los de Liesbeth De Mol que ha conseguido simplificar algunos sistemas tag usándolos para simular funciones de Collatz y algunos problemas de decidibilidad [3].

También, Turlough Neary encontrando máquinas de Turing universales cada vez más pequeñas aplicando sistemas tag en sus demostraciones, gracias a esto encontró la máquina de Turing universal de 3 colores y 11 estados, la de 6 colores y 6 estados entre otras, el uso de los sistemas tag implicó una reducción en costo computacional de sus demostraciones haciéndolas más eficientes [6].

1.5. Sistemas Tag Generalizados

En el 2018, Turlough Neary y Matthew Cook encontraron una generalización de sistemas tag, este modelo además de simplificar su aplicación también dio la apertura para configurar de otra manera sistemas tag ya existentes y simularlos en tiempo lineal [7].

Para su modelo utilizaron el concepto de ciclo P, pero a diferencia de un sistema tag cíclico ya no agregaría la cadena al final únicamente si el carácter es 1, sino que el alfabeto puede variar en número símbolos y cada símbolo puede agregar una cadena al final según el ciclo de la máquina. Así con un solo modelo incluyeron los dos ya existentes y expandir las posibles configuraciones.

1.6. Máquinas de Turing como Autómatas Celulares

En una investigación similar a la de este trabajo que se produjo en el 2017 por Sergio Eduardo Juárez Martínez y César Iván Manzano Mendoza en el trabajo termi-

nal titulado “Máquinas de Turing y el problema de la universalidad como sistemas dinámicos discretos”, establecieron un algoritmo que convierte una máquina de Turing dada en un autómata celular equivalente [13].

En este trabajo terminal se propone el uso del modelo de sistemas tag, así como los autómatas celulares unidimensionales para encontrar un algoritmo tal que, al introducir una configuración de sistema tag, devuelve un autómata celular que lo simule y caracterizar a los sistemas tag como sistemas dinámicos discretos.

1.7. Tabla Comparativa

Título	Autor	Año	Máquina que Simula	Herramienta disponible
A New Kind of Science	Stephen Wolfram	2002	Varias máquinas de computación	Si
Universality in Elementary Cellular Automata	Matthew Cook	2004	Sistema Tag en Autómata de la Regla 110	No
Máquinas de Turing y el problema de la universalidad como sistemas dinámicos discretos	Sergio Eduardo Juárez Martínez y César Iván Manzano Mendoza	2017	Máquinas de Turing con Autómatas Celulares	Si

Tabla 1.1: Tabla comparativa de trabajos similares

Capítulo 2

Marco Teórico

2.1. Sistemas Tag

Los sistemas de tag fueron introducidos por Emil Post a principios de la década de 1920 buscando representar las expresiones del $Principia\ Mathematica$ de Bertrand Russell y Alfred North Whitehead en transformaciones de cadenas. Pero fue más allá para lograr que un problema de lógica-matemática se convierta en un problema de computación porque, al tratar de formalizar las transformaciones teóricamente, llegó a lo que llamó él $problema\ de\ la\ etiqueta$. Los sistemas tag se definen como:

Un sistema tag T es una tupla (Σ, H, R, P) donde,

- Σ es un conjunto finito de símbolos.
- H es un conjunto finito de símbolos de parada, tales que $\Sigma \cap H = \emptyset$.
- R es un conjunto de reglas $R : \Sigma \to \{\Sigma \cup H\}^*$ con una regla para cada símbolo de Σ.
- P es un número de borrado $P \in N$ con $P \geq 1$.

La configuración de un sistema tag es una cadena de símbolos $W = w_1 w_2 ... w_n$ dónde $w_i \in \Sigma \cup H$. Cuando se aplica la regla de la forma $w_1 \to A_1 \in \{\Sigma \cup H\}^*$ para

w_1, se borra la subcadena $w_1w_2...w_P$ y se adjunta la cadena A_1 en la parte derecha de la cadena W. Así, pues, cada paso de un sistema tag es determinístico y se expresa como:

$$w_iw_2w_3...w_n \vdash w_{P+1}...w_nA_i$$

El sistema tag se detiene cuando $|W| < P$ o cuando el símbolo más a la izquierda de la cadena es un símbolo de parada ($w_1 \in H$).

Como ejemplo, el sistema tag que Emil Post encontró con este problema fue R : $0 \to 00, 1 \to 1101$ con número de borrado $P = 3$.

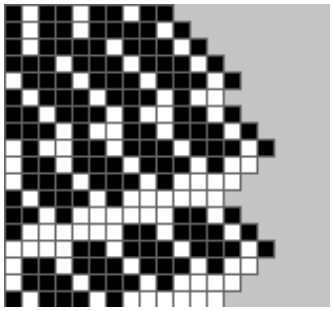


Figura 2.1: Ejemplo con la cadena inicial 1011011011.

2.2. Sistemas Tag Cíclicos

Los sistemas tag cíclicos fueron presentados por Matthew Cook en el 2004, fueron utilizados en su investigación sobre la universalidad de los autómatas celulares elementales, en específico de la regla 110. Es equivalente a un sistema tag, pero con un alfabeto binario $\{0, 1\}^*$, un número de borrado $P = 1$ y todas las cadenas adjuntas se añaden solo si el primer símbolo del lado izquierdo es 1. Se puede formalizar de la siguiente manera:

Un sistema tag cíclico C es una lista ordenada de cadenas $A_i \in \{0, 1\}^*$ llamadas adjuntas y un número β que se refiere a la cardinalidad del conjunto C.

Una configuración de un sistema tag cíclico es un par (k, W) donde k es el número que apunta a la cadena adjunta A_k y la cadena binaria $W = w_1w_2...w_n$. En cada transición se borra w_1, se calcula $k = ((k + 1)mod\beta)$ y, si $w_1 = 1$, se adjunta la cadena A_k en el lado derecho de la cadena W. Así, las transiciones del sistema tag cíclico se pueden expresar como:

$$(k, 1w_2w_3...w_n) \vdash (((k + 1)mod\beta), w_2w_3...w_nA_k)$$

$$(k, 0w_2w_3...w_n) \vdash (((k + 1)mod\beta), w_2w_3...w_n)$$

El sistema tag cíclico $C = \{0100, 10, \epsilon, 101010\}$ simula las iteraciones de la función de Collatz ($(3n + 1)/2$ si n es impar, $n/2$ si n es par) con cadenas de $(10)^n$.

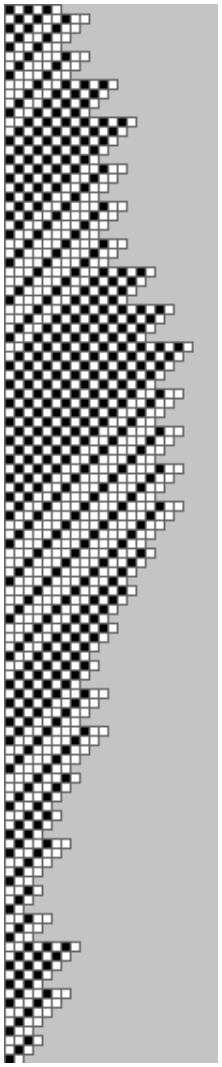


Figura 2.2: Ejemplo con la cadena inicial 101010 $((10)^3)$.

2.3. Sistemas Dinámicos Discretos

Un sistema dinámico es un conjunto de elementos que interactúan entre sí consiguiendo evolucionar con el tiempo. Los sistemas dinámicos discretos se pueden formalizar matemáticamente como:

$$x_{k+1} = f(x_k), \quad k = 0, 1, 2, 3, ...$$

Donde f es una función $f : X \to X$ definida en el espacio de estados X. Desde el siglo XX se han utilizado los métodos de estudio de sistemas dinámicos junto con los de análisis computacional en la investigación de modelos computacionales, algo que ha aumentado el conocimiento que se tiene de estos. Algunos de estos métodos son:

- Morfología Matemática
- Diagramas de Atractores
- Entropía de Shannon
- Diagramas de De Bruijn

2.4. Autómatas Celulares

Los autómatas celulares fueron ideados por John von Neumann en la segunda mitad del siglo XX como sistemas de computación paralela en los que él buscaba estudiar sistemas físicos. Estos sistemas computacionales realizan los cálculos basados en una rejilla donde cada celda representa un valor y cambia según el comportamiento de sus celdas vecinas. Formalmente, se pueden definir como:

Sea $A_C = (\Sigma, L, u, f)$ un autómata celular donde,

- Σ es un conjunto finito de estados posibles para cada celda.
- L es un arreglo de dimensión $d \in \mathbb{N}$, es decir, la rejilla del autómata celular.
- u es una vecindad donde se relacionan localmente las celdas de la rejilla de manera que, $u : L \to L^n$, $n \in \mathbb{N}$.
- f es una función de transición definida como $f : \Sigma^n \to \Sigma$.

El cual actualiza el estado de cada celda $x \in \Sigma$ en un tiempo discreto t haciendo uso de la regla $x^{t+1} = f(u(x)^t)$ de manera simultánea a las demás. Una transición global G del A_C se expresa como $G : \Sigma^L \rightarrow \Sigma^L$ que mapea cualquier estado global o configuración c^t a otra c^{t+1}, $G(c^t) = c^{t+1}$. Así las configuraciones del A_C desde la configuración inicial se pueden expresar como:

$$c^0 \vdash c^1 \vdash ... \vdash c^t \vdash c^{t+1}$$

Convencionalmente en un A_C una vecindad se define según un radio r de celdas que se encuentren alrededor de una celda. Ejemplos de esto son los *Automatas Celulares Elementales* de Stephen Wolfram y el *Juego de la Vida* de John Conway. Donde se usa $r = 1$ en ambos casos, es decir, la vecindad es un conjunto de celdas donde cada celda vecina esta solo a una posición de distancia de la celda central.

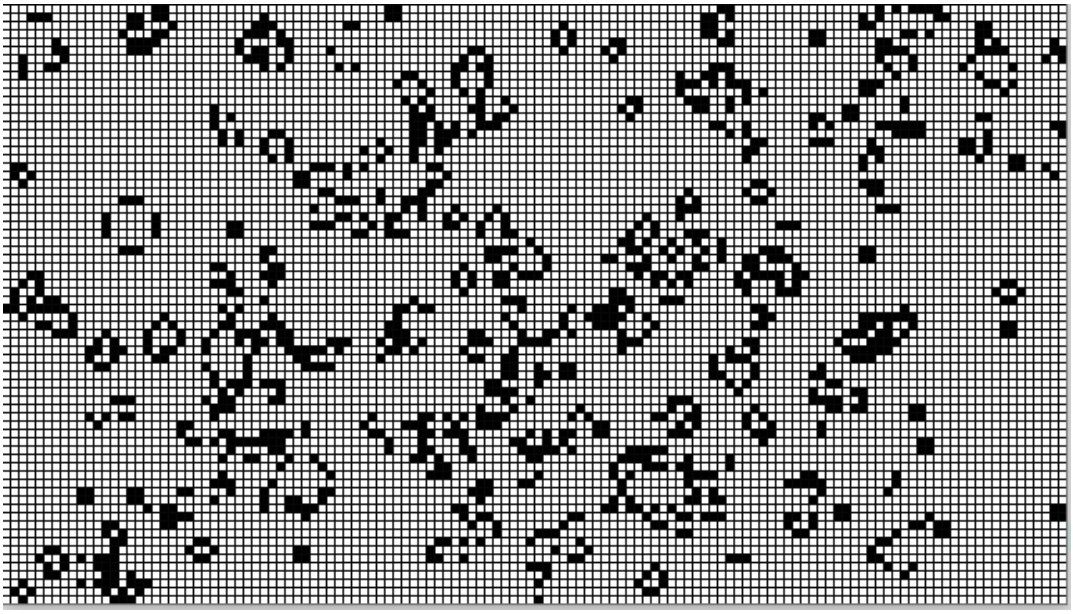


Figura 2.3: Ejemplo de simulación del *Juego de la Vida* de John Conway.

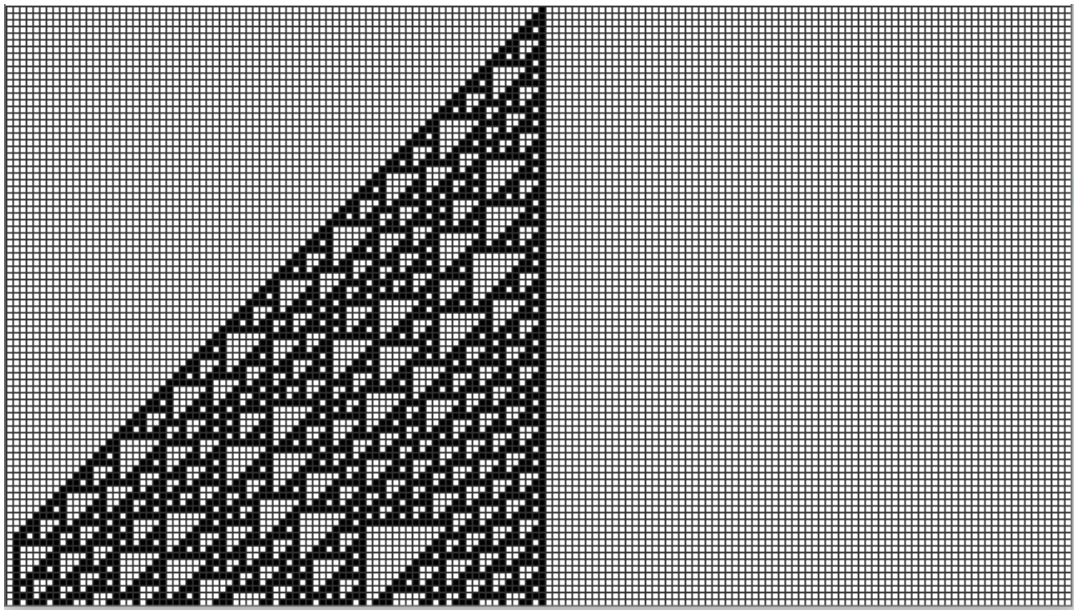


Figura 2.4: Ejemplo de simulación de la regla 110, un *Autómata Celular Elemental* de Stephen Wolfram.

Capítulo 3

Planteamiento del Problema

El estudio de modelos computaciones como sistemas dinámicos discretos comenzó a ser tendencia en los últimos años, la principal característica de estos es que cambian con el tiempo y muchos modelos de computación tienen esta característica porque al aplicar las reglas del modelo en los datos de entrada estos van cambiando por cada transición o iteración hasta llegar a un resultado final o de salida como los aplicó Wolfram en [11]. Otra cosa que caracteriza a los sistemas dinámicos es que sus elementos interactúan unos con otros y en este sentido algunos modelos computacionales no cumplen con esta característica totalmente como los sistemas tag que tienen reglas logísticas en donde no hay una relación evidente entre los elementos de una cadena.

Hasta el momento los sistemas tag son estudiados en su aplicación como programas para calcular funciones y simulando otras máquinas, usualmente el análisis se hace con formalismos matemáticos como en [3][6]. En este trabajo terminal se emplearán métodos sistemáticos que se utilizan para el estudio de sistemas dinámicos discretos y así incluir a los sistemas tag entre los modelos que son estudiados de esta forma.

La hipótesis principal es que existe un algoritmo que convierta la configuración de

un sistema tag en un autómata celular aprovechando su potencial de computación. En la búsqueda de este algoritmo se quieren resolver las siguientes preguntas:

- ¿Qué configuraciones de sistemas tag se pueden convertir en reglas de autómatas celulares?
- ¿Cómo sería el proceso de conversión?
- ¿Qué características se pueden obtener del análisis de sistemas tag como sistemas dinámicos discretos?
- ¿Qué características tienen los autómatas celulares que simulan sistemas tag?

Con base en estas cuestiones los objetivos de esta investigación aparecen a continuación.

3.1. Objetivos

3.1.1. Objetivo General

Analizar sistemas tag como sistemas dinámicos con su simulación en autómatas celulares.

3.1.2. Objetivos Específicos

- Desarrollar una herramienta para la simulación de la dinámica espacial de los sistemas tag.
- Analizar el comportamiento de los sistemas tag en su dinámica espacial.
- Analizar autómatas celulares unidimensionales que simulen sistemas tag.
- Contribuir al estudio de los sistemas tag como sistemas dinámicos.

3.2. Justificación

La herramienta computacional con la que se estudiarán los sistemas tag como sistemas dinámicos son los autómatas celulares. Estos no han tenido tantas aplicaciones en la industria tecnológica, su uso común se relaciona con el ámbito científico y de investigación de otras áreas aparte de la computación como los sistemas complejos gracias a que pueden simular sistemas dinámicos con modelado discreto y su comportamiento se puede analizar desde varias perspectivas fenotípicas, genotípicas y morfológicas como se muestra en [14]. Gracias a la variedad de métodos que se tienen en la actualidad y su simplicidad, pueden cumplir con los requerimientos de esta investigación. Algunos de estos son:

- Morfología Matemática
- Diagramas de Atractores
- Entropía de Shannon
- Diagramas de De Bruijn

Hasta el momento no se han relacionado de esta manera a los sistemas tag con los autómatas celulares por lo que en este trabajo terminal se busca presentar esta relación y la caracterización en su dinámica espacial acrecentando el conocimiento de estos al embozar algún nuevo método para su estudio que puede ayudar a la comunidad científica y estudiantil del área para futuras aplicaciones e investigaciones y proveyendo de un simulador como herramienta. Pero, aunque se tengan varias formas de analizar un sistema dinámico discreto esta tarea no será trivial por tres razones:

- Se trata de aumentar el conocimiento de los sistemas tag que hasta el momento es subestimado y no ha sido tan basto como con otros modelos, pero ha mejorado varios aspectos de la computación [2][3].

- No hay hasta el momento un método para encontrar una configuración en autómatas celulares que pueda simularlos espacialmente.
- La herramienta propuesta como simulador de sistemas tag y su dinámica espacial en autómatas celulares debe optimizar la simulación cuando la demanda en memoria o tiempo de computación sean altos además debe ser fácil de utilizar.

La motivación principal para este trabajo terminal es aportar al estudio de los sistemas tag porque su universalidad computacional ha quedado demostrada en varios trabajos como [1][2][6][10].

La dificultad que se presenta no es despreciable los métodos para el desarrollo de la investigación ya fueron previamente estudiados en las unidades de aprendizaje de la carrera en curso.

Capítulo 4

Sistemas Tag y Autómatas Celulares

4.1. Autómatas Celulares Unidimensionales

Los primeros autómatas celulares habían sido bidimensionales, es decir, que estaban definidos en un espacio bidimensional. En los autómatas celulares unidimensionales la celda central es influenciada por las celdas laterales respecto a un rango constante y definido r, este parámetro suele ser entero en un espacio discreto, pero es posible que sea fraccionario. Si el alfabeto del autómata celular es $\Sigma = \{\sigma_1, ..., \sigma_n\}$ la representación de la regla de transición es:

$$w_{i-r}...w_{i-2}w_{i-1}w_iw_{i+1}w_{i+2}...w_{i+r} \rightarrow \sigma_k$$

w_{i-r}	...	w_{i-2}	w_{i-1}	w_i	w_{i+1}	w_{i+2}	...	w_{i+r}
				σ_k				

Tabla 4.1: Vecindad de un autómata celular unidimensional.

y el número de configuraciones posibles se calcula como:

$$C_T = |\Sigma|^{2r+1}$$

Como consecuencia de esto mientras más rango tenga el autómata celular hay un número mayor de configuraciones. Y el número de reglas posibles estará dado por:

$$R_T = |\Sigma|^{C_T}$$

Si se toma en cuenta un alfabeto binario $\Sigma = \{0, 1\}$ y un radio $r = 2$ se obtienen:

$$C_T = 2^{2(2)+1} = 32$$

$$R_T = 2^{32} = 4\ 294\ 967\ 296$$

4.2. Autómatas Celulares Elementales

Los autómatas celulares elementales son autómatas celulares unidimensionales de alfabeto $\Sigma = \{0, 1\}$ y radio $r = 1$. Por tanto, las posibles configuraciones y reglas son:

$$C_T = 2^{2(1)+1} = 8$$

$$R_T = 2^8 = 256$$

Como el alfabeto es binario se puede utilizar un código que simplifica la definición de un autómata celular elemental por la regla que utiliza es decir, según el estado de las dos celdas que tiene a cada lado y su estado actual, las posibles configuraciones son: 111, 110, 101, 100, 011, 010, 001, 000 en ese orden y el siguiente estado de la celda central en cada una de estas se da con la representación binaria de un número entre el 0 y el 255 como el $110_d = 01101110_b$ cuya configuración es:

111	→	0
110	→	1
101	→	1
100	→	0
011	→	1
010	→	1
001	→	1
000	→	0

Tabla 4.2: Reglas de transición del autómata celular elemental de la regla 110.

Así, este autómata celular elemental es llamado el autómata de la regla 110 y a continuación se muestra una simulación de dicho autómata con una cadena de entrada aleatoria.

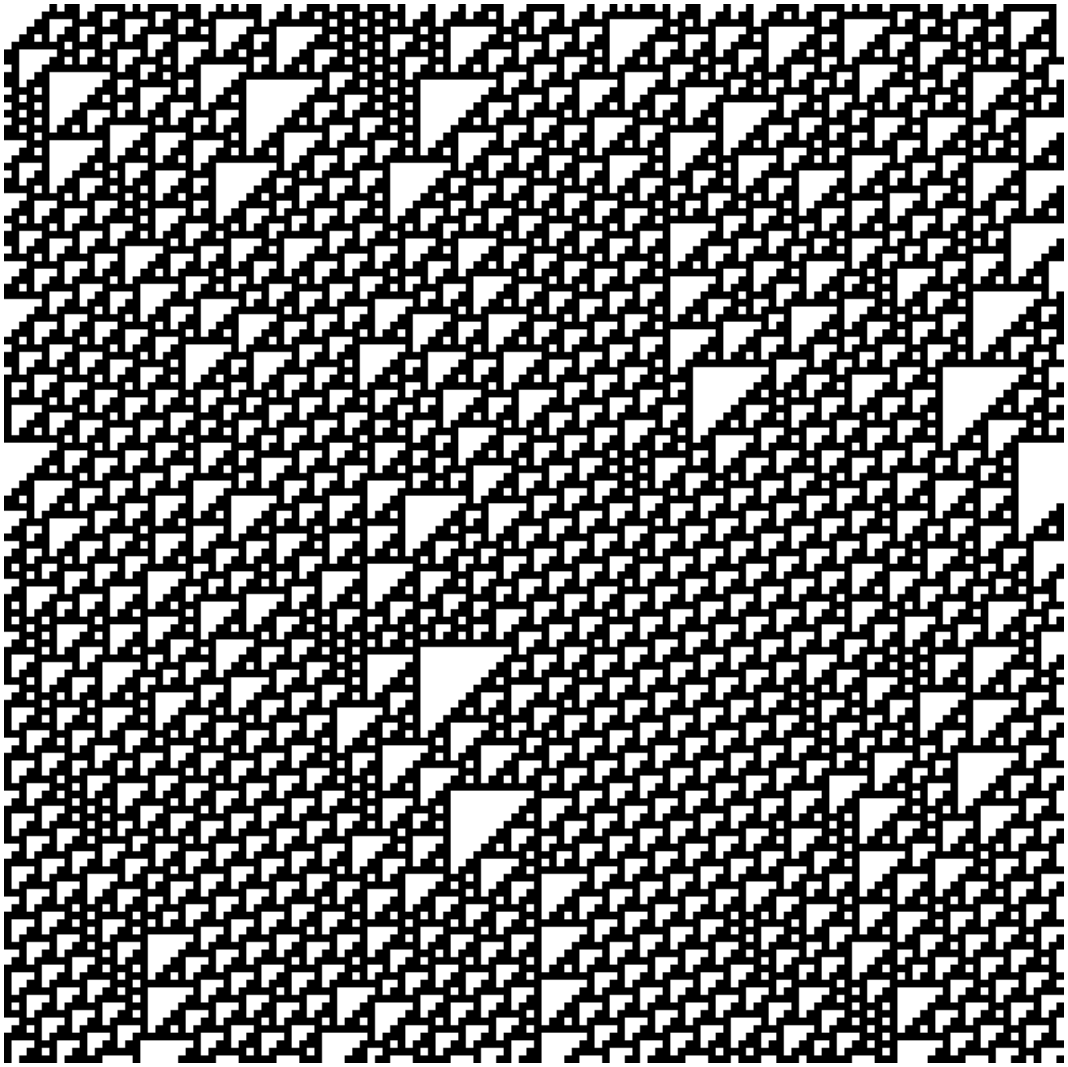


Figura 4.1: Simulación del autómata celular de la regla 110 con una cadena inicial aleatoria

Por su comportamiento complejo se ha encontrado que el autómata celular elemental de la regla 110 es capaz de hacer computación universal, la demostración fue dada por Matthew Cook en la cual simula dentro del autómata un sistema tag universal [2].

4.3. Clasificación de los Autómatas Celulares

El estudio del comportamiento de los autómatas celulares ha llevado a plantear diferentes formas de clasificarlos. La clasificación de Wolfram ha sido más utilizada por su descripción basada en la complejidad y comportamiento. Consta de cuatro clases.

- Clase I. Evolución a un estado uniforme. Se le asigna esta clase a los autómatas celulares que convergen a un estado homogéneo y ordenado después de unas cuantas evoluciones.

Figura 4.2: Simulación del autómata celular de la regla 40 que pertenece a la clase I con una cadena inicial aleatoria

- Clase II. Evolución a estados periódicos. Caracteriza a autómatas celulares que

presentan una evolución cíclica, es decir, que a través de un número constante (periodo) de estados se repiten.

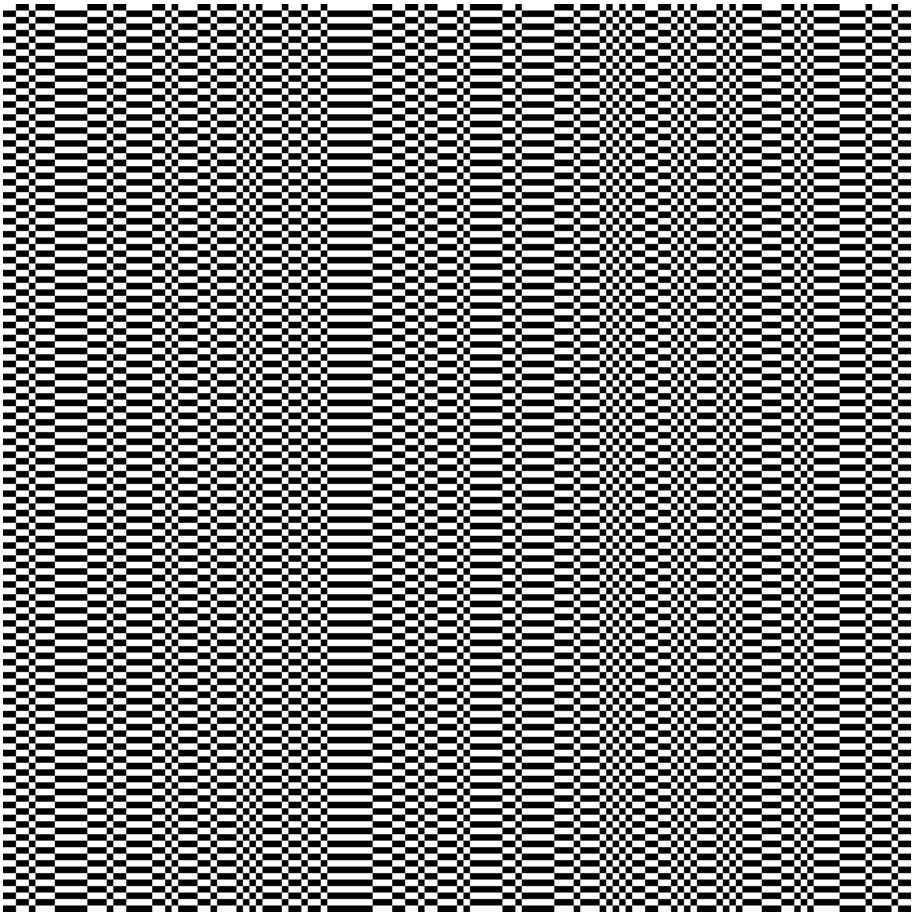


Figura 4.3: Simulación del autómata celular de la regla 51 que pertenece a la clase II con una cadena inicial aleatoria

- Clase III. Evolución caótica. Está caracterizada por un comportamiento aleatorio en el cual aparecen patrones periódicos aislados que forman triángulos.

Figura 4.4: Simulación del autómata celular de la regla 30 que pertenece a la clase III con una cadena inicial aleatoria

- Clase IV. Evolución con un comportamiento complejo. Los autómatas celulares que pertenecen a esta clase presentan un comportamiento que incluye a todos los anteriores de manera que emergen patrones complejos.

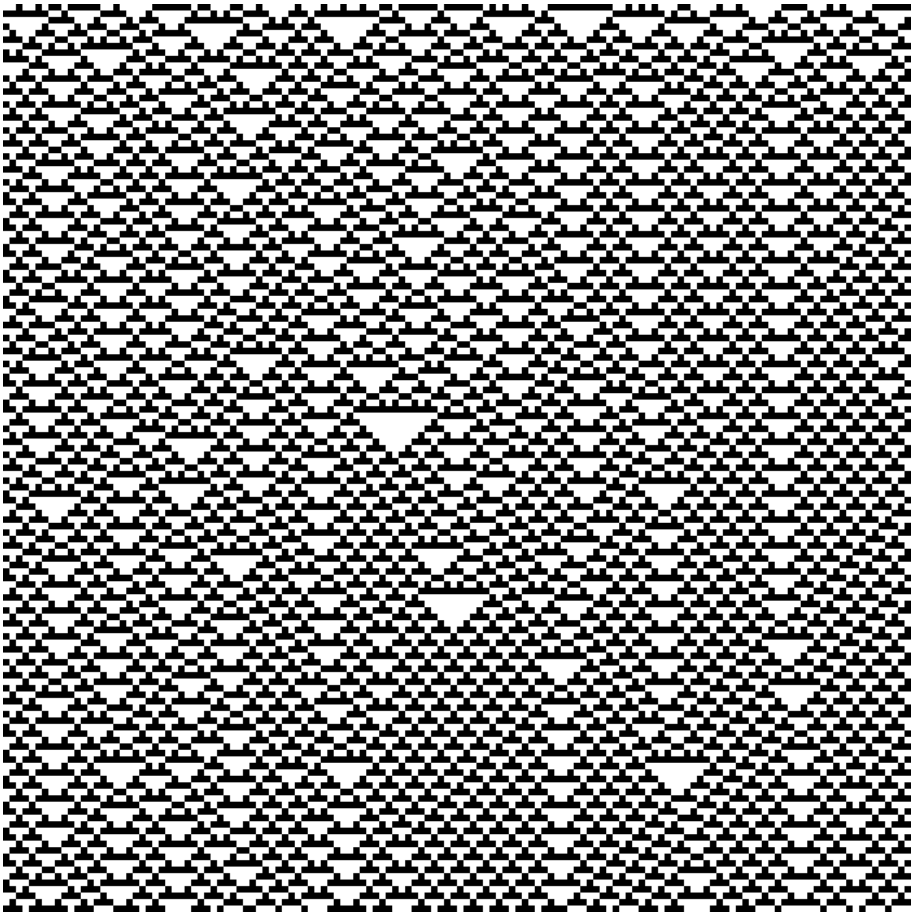


Figura 4.5: Simulación del autómata celular de la regla 54 que pertenece a la clase IV con una cadena inicial aleatoria

Estas clases están enumeradas de manera ascendente según el aumento de la complejidad, el autómata celular elemental universal de la regla 110 pertenece a la clase IV, hecho que relaciona el comportamiento complejo con la universalidad de los sistemas.

En esta clasificación se pueden incluir toda clase de autómatas celulares porque provee de una clasificación simple y fenotípica, es decir, que se puede percibir de manera gráfica.

4.4. Funcionamiento de un Sistema Tag

En la práctica, al definir un sistema tag, basta con proporcionar las reglas de producción y el número de borrado.

Por ejemplo, el sistema tag que genera cadenas de Fibonacci. Las cadenas de Fibonacci, como la sucesión de Fibonacci, genera una palabra concatenando las dos palabras anteriores mediante:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 01 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Hasta $n = 4$ las cadenas de Fibonacci son las siguientes:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 01 \\ f(2) &= f(1) + f(0) = 010 \\ f(3) &= f(2) + f(1) = 01001 \\ f(4) &= f(3) + f(2) = 01001010 \end{aligned}$$

Para poder generar las cadenas de Fibonacci se utiliza el sistema tag [11]

$$0 \to 01, 1 \to 0$$

$$P = 1$$

Cuya definición se describe como: cada vez que encuentre un 0 en el extremo izquierdo de la cadena se agrega la cadena 01 en el extremo derecho y cuando encuen-

tre un 1 se agrega la cadena 0, por cada transición se borra 1 carácter del extremo izquierdo de la cadena. Para mostrar este funcionamiento se introduce la cadena 0 con la que después de 12 pasos de computación se obtiene $f(4)$.

$$\mathbf{0} \vdash \mathbf{01} \vdash 101 \vdash \mathbf{010} \vdash 1001 \vdash 0010 \vdash \mathbf{01001}$$

$$\vdash 100101 \vdash 001010 \vdash 0101001 \vdash 10100101 \vdash \mathbf{01001010}$$

4.5. Visualización Espacial de los Sistemas Tag

Definida la manera de funcionar de un sistema tag se puede observar de manera visual y analizar los patrones que surgen a través de sus transiciones. Se propone que se tome cada cadena generada por cada transición y se ubiquen cada una debajo de la otra. Tomando el ejemplo del sistema tag que genera cadenas de Fibonacci se obtiene:

0								
0	**1**							
1	0	1						
0	**1**	**0**						
1	0	0	1					
0	0	1	0					
0	**1**	**0**	**0**	**1**				
1	0	0	1	0	1			
0	0	1	0	1	0			
0	1	0	1	0	0	1		
1	0	1	0	0	1	0	1	
0	**1**	**0**	**0**	**1**	**0**	**1**	**0**	

Tabla 4.3: Primeras 12 transiciones del sistema tag $0 \to 01, 1 \to 0;\ P = 1$.

Como cada cadena ocupa un espacio unidimensional y la sucesión de estas, una debajo de la otra, se vuelve bidimensional entonces se utiliza un plano bidimensional para verlo gráficamente asignando a cada carácter un color como el blanco para los

0s y el negro para los 1s. Se obtiene la siguiente figura:

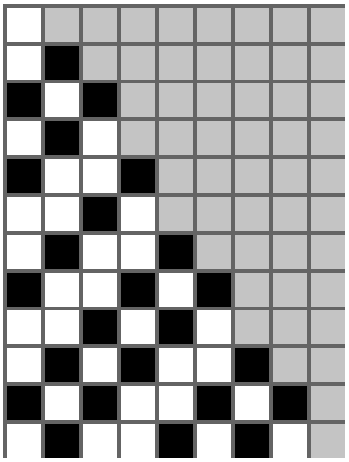


Figura 4.6: Visualización gráfica de los primeros 12 pasos del sistema tag $0 \to 01, 1 \to 0$; $P = 1$ con la cadena inicial 0.

Así es posible obtener una descripción de las instantáneas del sistema por cada paso o evolución de este. Este método fue promovido principalmente por Wolfram en su libro *A New Kind of Science* [11].

4.6. Sistemas Tag y la Función de Collatz

Existen algunos sistemas tag que pueden simular funciones matemáticas de manera sencilla. Tal es el caso de la función de Collatz definida de esta forma:

$$f(n) = \begin{cases} \frac{3n+1}{2}, & \text{si } n \text{ es impar} \\ \\ \frac{n}{2}, & \text{si } n \text{ es par} \end{cases}$$

Esta función es recursiva, es decir, $f(n_{i+1}) = f(n_i)$ hasta que tienda a 1. Para $n = 7$ obtenemos la serie:

$$f(7) = 11, f(11) = 17, f(17) = 26, f(26) = 13, f(13) = 20, f(20) = 10,$$
$$f(10) = 5, f(5) = 8, f(8) = 4, f(4) = 2, f(2) = 1$$

Esta función puede ser simulada con el sistema tag $0 \rightarrow 12, 1 \rightarrow 0, 2 \rightarrow 000$ con número de borrado $P = 2$. En el cual al introducir una cadena de la forma 0^n esta producirá las cadenas donde se cumpla la función $f(n)$. A continuación un ejemplo con la cadena $0000000 = 0^7$.

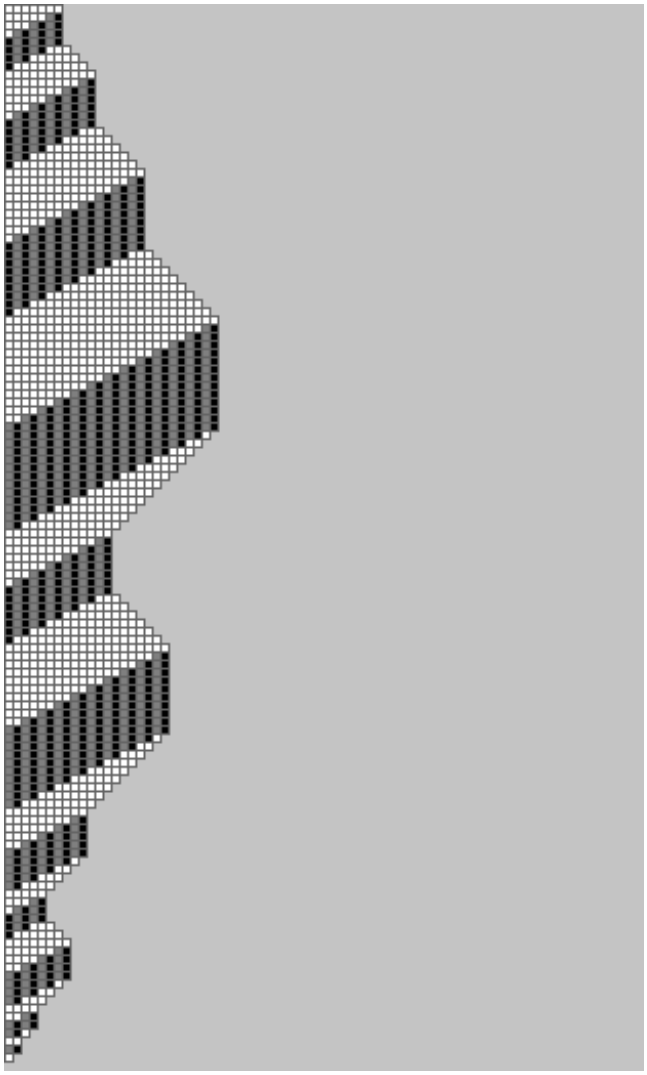


Figura 4.7: Sistema tag $0 \to 12, 1 \to 0, 2 \to 000$ con la cadena inicial $0000000 = 0^7$ en 128 evoluciones.

Gracias a la visualización espacial se nota la evolución en tiempo lineal $O(n)$ que genera cada cadena 0^n. Este sistema tag fue diseñado por De Mol en 2008 [3] donde provee de una forma de encontrar sistemas tag que simulen la familia de funciones de Collatz.

4.7. Sistema Tag que simula una Máquina de Turing Universal

Una de las máquinas de Turing universales más pequeñas es la que tiene dos estados y tres colores, sugerida por Wolfram en 2002 [11] y efectuada la demostración de su universalidad por Alex Smith en 2007 [18] para lo cual utilizó sistemas tag cíclicos.

	A	B
0	P1,R,B	P2,L,A
1	P2,L,A	P2,R,B
2	P1,L,A	P0,R,A

Tabla 4.4: Transiciones de la máquina de Turing universal de Wolfram[11].

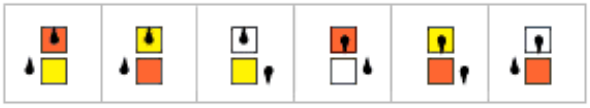


Figura 4.8: Máquina de Turing Universal de dos estados y tres colores. [11]

El sistema tag que se define a continuación puede simular la máquina de Turing universal simple (de dos estados y tres colores) [11].

$$0 \to 32, 1 \to 65, 2 \to 11, 3 \to 0, 4 \to, 5 \to 00, 6 \to 11, 7 \to 00, 8 \to ba, 9 \to ed, a \to 9999,$$
$$b \to 8, c \to 8, d \to 8, e \to 9999, f \to 8888, g \to ji, h \to ml, i \to 99hh, j \to ggoooo, k \to ggoo,$$
$$l \to ggoooo, m \to 9999h, n \to, o \to rq, p \to ut, q \to p, r \to oooo, s \to oooo, t \to oooo,$$
$$u \to p, v \to o; \quad P = 2$$

Este sistema tag produce los movimientos del cabezal de la máquina de Turing de

Wolfram. Porque los movimientos del cabezal son una de las principales características de esta máquina.

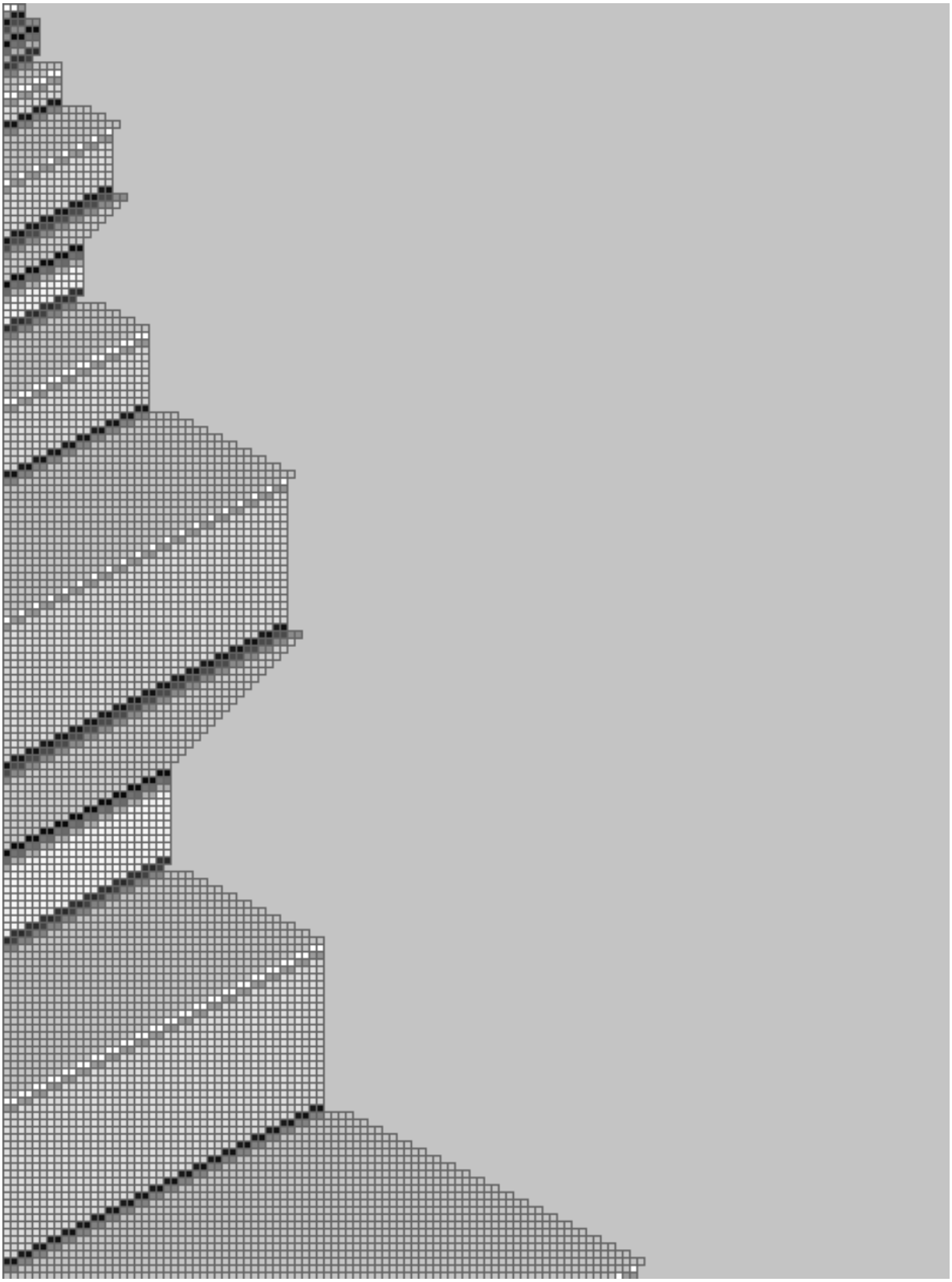


Figura 4.9: Sistema tag que simula la máquina de Turing universal de dos estados y tres colores con la cadena inicial $00i$ en 175 evoluciones.

4.8. El Problema de la Etiqueta

Cuando Emil Post ideó sus sistemas tag se encontró con uno en particular que le causó mucho tiempo de análisis. Este sistema de alfabeto binario $\Sigma = \{0, 1\}$ número de borrado $P = 3$ y las reglas de producción $0 \to 00, 1 \to 1101$ cuyo comportamiento le hacía preguntarse si para algunas cadenas este no pararía sino que seguiría infinitamente, sin que necesariamente entre en un bucle [8].

Stephen Wolfram retomó la investigación de Emil Post para comprobar su *Principio de Equivalencia Computacional* que significa que *todos los procesos, ya sean producidos por el esfuerzo humano o que ocurren espontáneamente en la naturaleza, pueden verse como computaciones* que pueden reproducirse en los sistemas computacionales más simples (por ejemplo las máquinas de Turing o los autómatas celulares) y existe una equivalencia entre ellos. Aunque aún no ha llegado a la conclusión satisfactoria sobre los sistemas tag simples, ha utilizado una metodología sistemática para el análisis en la cual busca la reducción de los problemas y la simulación de las transiciones de cada configuración.

Si tomamos el sistema tag anteriormente definido con una cadena aleatoria, se produce una simulación visual de sus configuraciones por cada transición poniendo una debajo de la otra se obtiene su evolución en un espacio bidimensional con la cual es posible saber de manera gráfica si existe un patrón por cada paso de computación. En la figura 4.10 se encuentra un ejemplo con la cadena inicial aleatoria de longitud 65.

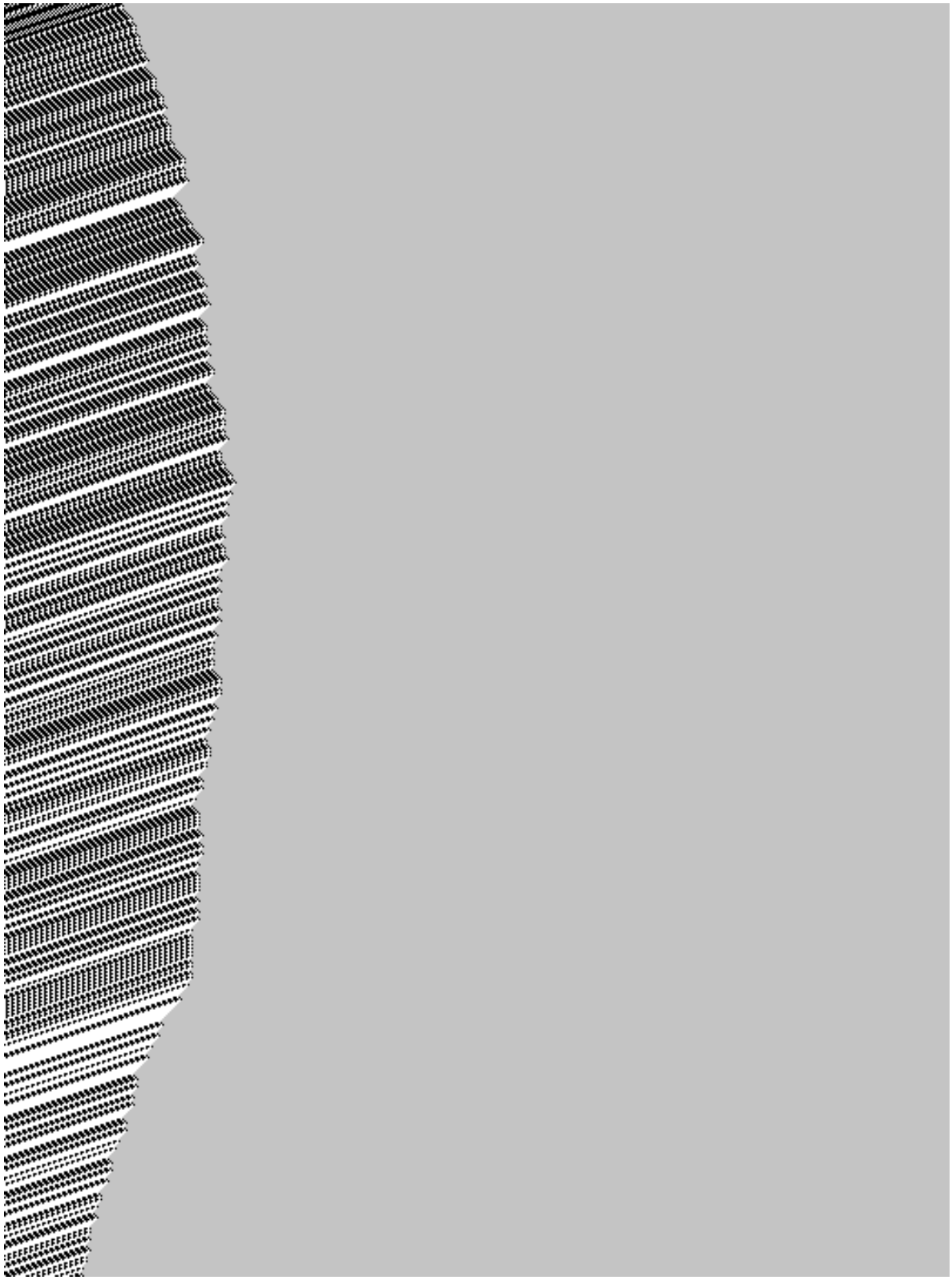


Figura 4.10: Simulación del sistema tag $0 \to 00, 1 \to 1101$ con una cadena inicial aleatoria en 700 evoluciones.

Como se puede observar para que el sistema pare pueden necesitarse un número más extenso de pasos el cual puede tener el orden de los millones según las aportaciones de Stephen Wolfram.

El sistema tag $0 \rightarrow 00, 1 \rightarrow 1101$ todavía es estudiado debido a que aún no se ha encontrado una cadena con la que este siga calculando de manera infinita aunque se conjetura que existe, el análisis de esta sección no se enfocará en encontrar un método para obtener esa cadena sino en las características del sistema.

4.9. Análisis sobre los Sistemas Tag

Al haber probado algunos sistemas tag con características variadas y de diferentes orígenes, es posible observar algunas cosas referentes al futuro diseño de autómatas celulares equivalentes. Los dos mecanismos fundamentales de un sistema tag son: el borrado y la concatenación de la cadena adjunta mediante la verificación del primer carácter de la cadena.

De manera fenotípica se observan patrones los sistemas tag tales como corrimientos hacia la izquierda dependientes del número de borrado, ciclos y bloques de subcadenas que tienen algún patrón de símbolos repetitivos.

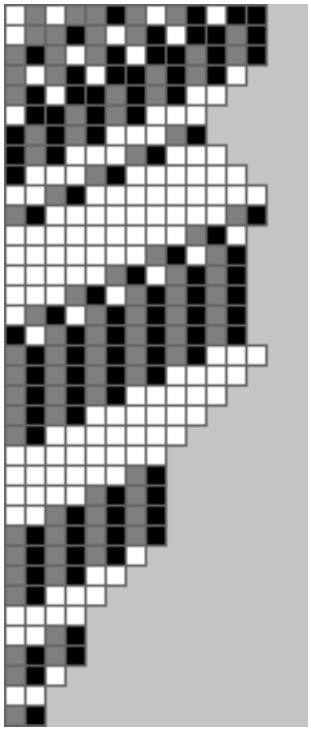


Figura 4.11: Simulación del sistema tag de la sección 4.7, se notan los corrimientos y los bloques de subcadenas con patrones.

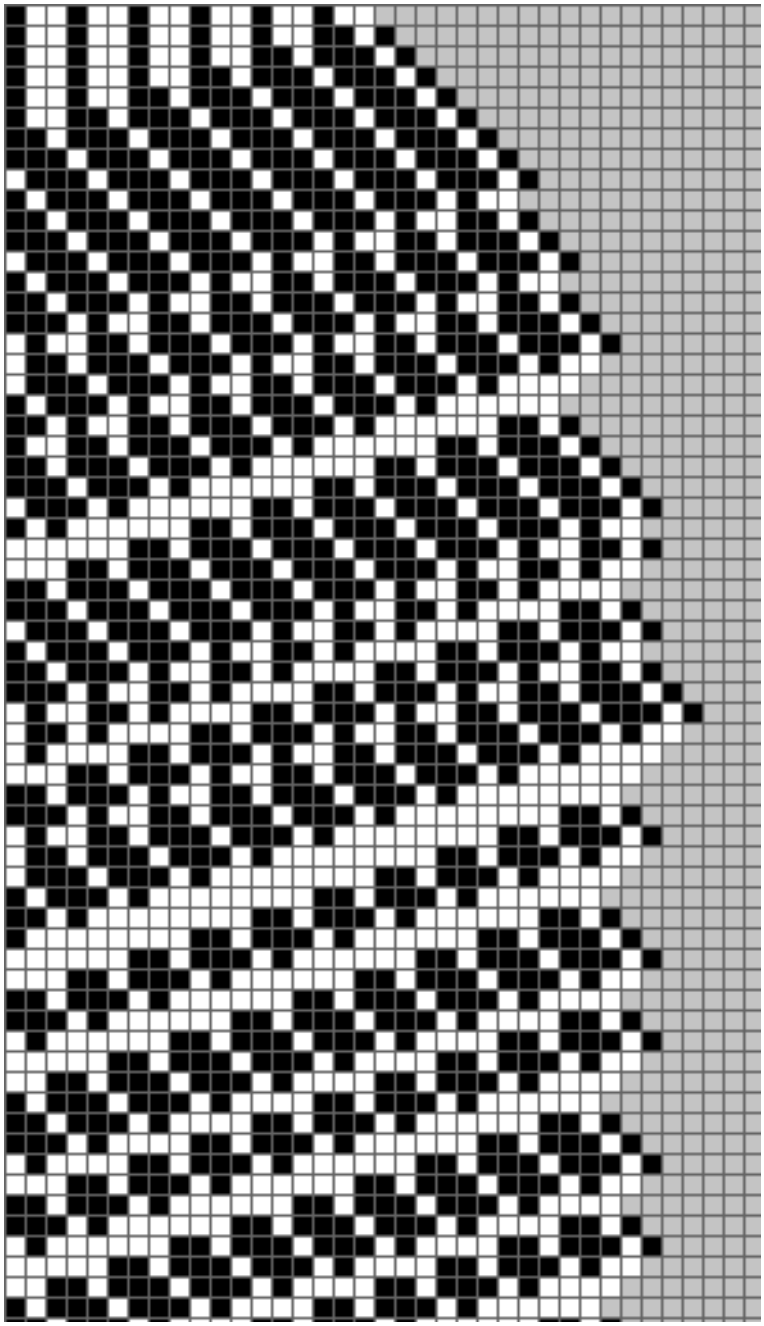


Figura 4.12: Simulación del sistema tag de la sección 4.8 que converge a un ciclo.

Aunque es de primera instancia según lo que se observa que existe un problema abierto a resolver: ¿se pueden encontrar las reglas de transición de un autómata celular unidimensional que simulen dicho comportamiento? Una primera respuesta a esta pregunta es: no. Debido a que la naturaleza del sistema tag viene de la verifi-

cación del primer carácter que encuentra en el extremo izquierdo para decidir qué cadena se adjuntará en el extremo derecho y un autómata celular tiene un radio especifico y constante en el que se define la vecindad de este y, en consecuencia, sus reglas de transición y se desenvuelven en un espacio infinito de celdas así que no puede verificar el extremo izquierdo de la cadena cuando esta tenga una extensión mayor al radio. Por lo tanto, es indeterminado que una regla de transición simule el comportamiento de la regla de producción o el patrón fenotípico del sistema tag.

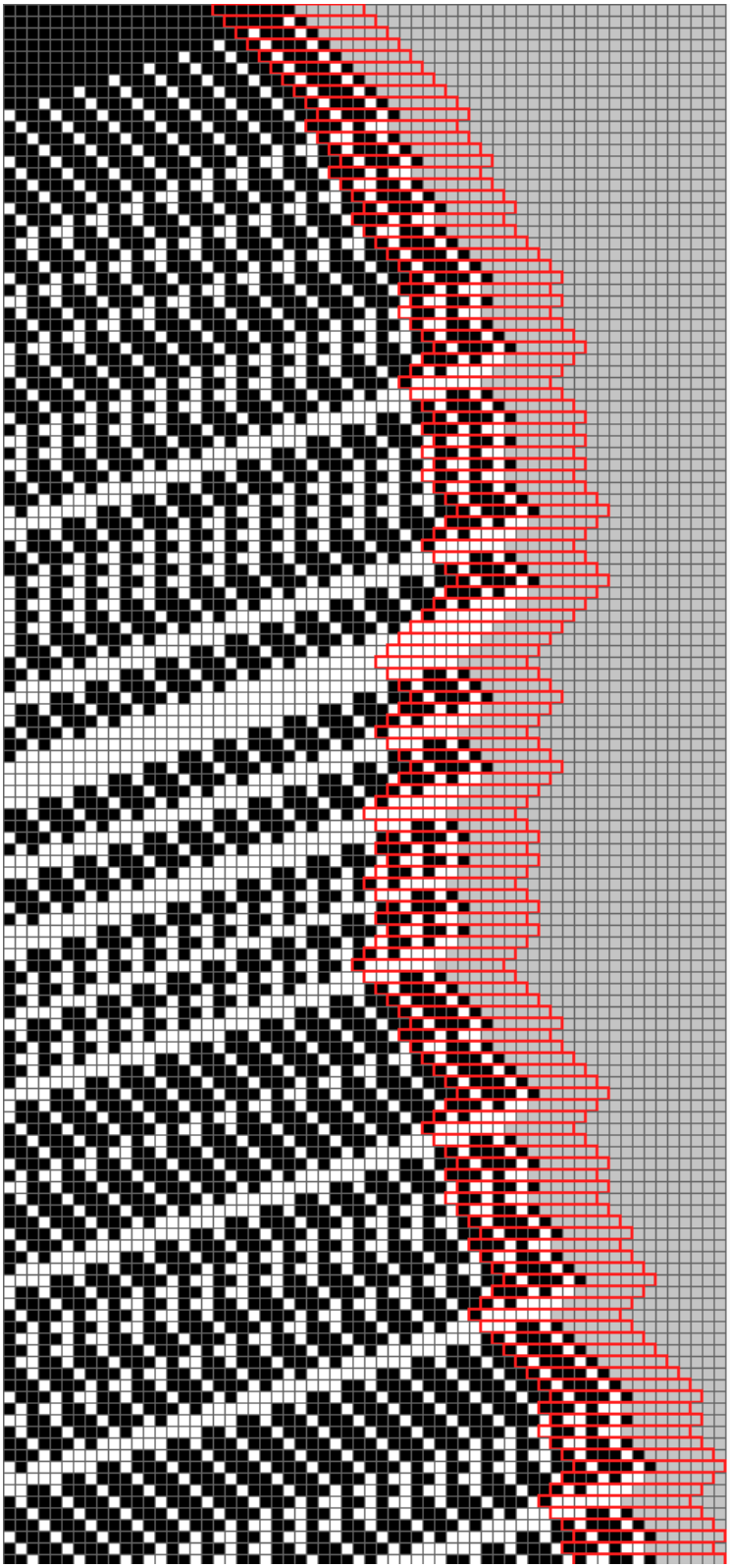


Figura 4.13: Simulación del sistema tag de la sección 4.8, se nota que no hay un patrón determinado al final de cada cadena generada por este.

Debido a esta observación sería difícil encontrar un autómata celular que equivalga de manera directa al patrón fenotípico de un sistema tag, pero esa no es la única manera de resolver el problema de la conversión. Se podría establecer un mecanismo expresado en las reglas de transición del autómata celular que permita borrar y añadir las cadenas adjuntas y en ese caso se obtendría cada cadena del sistema tag dada una cadena inicial, pero habría pasos intermedios de tal manera que el autómata celular manipule la cadena con dicho mecanismo para obtener la siguiente y este mecanismo debería estar bien conformado para que en cualquier caso realice la tarea en cuestión.

Como se vio en la sección 4.1 el radio en el que se define la vecindad de un autómata celular aumenta exponencialmente el número de configuraciones posibles C_T y, en consecuencia, el número de reglas posibles R_T. El número de símbolos en el alfabeto realmente no tiene un impacto exponencial sino lineal en esto. Así como parte un funcionamiento óptimo es necesario establecer el menor radio posible, aunque signifique aumentar el número de símbolos en el alfabeto de este, incluso se conjetura que un autómata celular con vecindad de radio $r = 1$ es suficiente aunque el número de símbolos del alfabeto será diferente en cada caso.

Ahora bien, un sistema tag no se desenvuelve en un espacio de celdas infinito como el autómata celular, de hecho, la longitud de cada cadena es variable a través del proceso así que para permitir la compatibilidad en cuanto al espacio infinito del autómata celular se añadirá un símbolo al alfabeto de este porque el alfabeto autómata celular contendrá el alfabeto del sistema tag y además un símbolo que represente al espacio vacío o nulo para abarcar el espacio infinito del autómata celular y extenderse a los laterales de cada cadena del sistema tag y de todas las cadenas intermedias delimitando el inicio y final de estas.

Capítulo 5

Autómatas Celulares que Simulan Sistemas Tag

5.1. Sistema Tag de la Sucesión de Thue-Morse

La sucesión de Thue-Morse es una sucesión de dígitos binarios que se obtiene al concatenar el complemento booleano de cada cadena empezando desde el cero. Se puede plantear de la siguiente forma:

$$T_0 = 0$$

$$T_n = T_{n-1}\neg T_{n-1}$$

La sucesión hasta $n = 6$ es:

$T_0 = 0$

$T_1 = 01$

$T_2 = 0110$

$T_3 = 01101001$

$T_4 = 0110100110010110$

$T_5 = 01101001100101101001011001101001$

$T_6 = 0110100110010110100101100110100110010110011010010110100110010110$

Se nota que la cadena crece de manera exponencial a razón de 2^n. El sistema tag que simula dicha sucesión es [11] :

$$0 \to 01, 1 \to 10$$

$$P = 1$$

Las primeras transiciones de este sistema tag hasta que generan T_4 son:

0															
0	**1**														
1	0	1													
0	**1**	**1**	**0**												
1	1	0	0	1											
1	0	0	1	1	0										
0	0	1	1	0	1	0									
0	**1**	**1**	**0**	**1**	**0**	**0**	**1**								
1	1	0	1	0	0	1	0	1							
1	0	1	0	0	1	0	1	1	0						
0	1	0	0	1	0	1	1	0	1	0					
1	0	0	1	0	1	1	0	1	0	0	1				
0	0	1	0	1	1	0	1	0	0	1	1	0			
0	1	0	1	1	0	1	0	0	1	1	0	0	1		
1	0	1	1	0	1	0	0	1	1	0	0	1	0	1	
0	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**

Tabla 5.1: Primeras 16 transiciones del sistema tag de la sucesión de Thue-Morse con la cadena inicial 0.

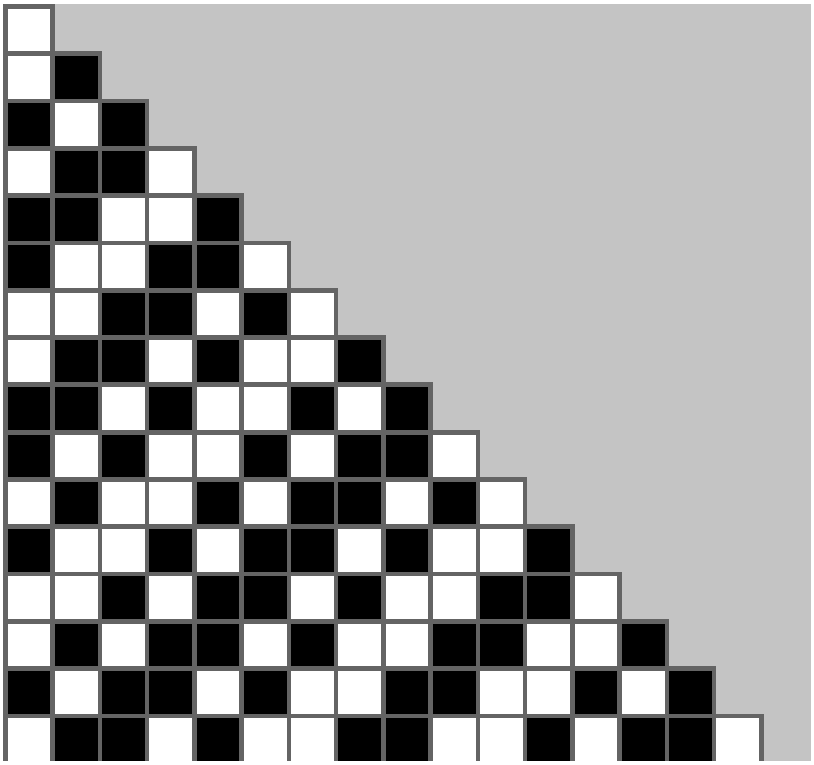


Figura 5.1: Visualización de las primeras 16 evoluciones del sistema tag $0 \rightarrow 01, 1 \rightarrow 10; P = 1$

Este sistema tag necesita de 2^n transiciones para llegar a la cadena T_n y tiene una definición simple, así que se puede proponer un autómata celular que lo simule.

Como cada transición del sistema tag producen una cadena unidimensional, un autómata celular unidimensional será óptimo para simularlo. Aunque es necesario realizar un análisis de los patrones que se encuentran en el sistema tag.

5.1.1. Análisis del Sistema Tag de la Sucesión de Thue-Morse

De manera fenotípica se observa un patrón en el sistema tag, gracias a que su número de borrado es $P = 1$ se nota un corrimiento a la izquierda de un carácter por transición y aumenta uno a la izquierda.

Aunque es de primera instancia según lo que se observa que existe un problema

abierto a resolver: ¿se pueden encontrar las reglas de transición de un autómata celular unidimensional que simulen dicho comportamiento? Una primera respuesta a esta pregunta es: no. Debido a que la naturaleza del sistema tag viene de la verificación del primer carácter que encuentra en el extremo izquierdo para decidir que cadena se adjuntará en el extremo derecho y un autómata celular tiene un rango específico y constante en el que se basan sus reglas de transición y se desenvuelven en un espacio infinito de celdas así que no puede verificar el extremo izquierdo de la cadena cuando esta tenga una extensión mayor al radio. Por lo tanto, es indeterminado que una regla de transición simule el comportamiento de la regla de producción o el patrón fenotípico del sistema tag.

Cabe señalar que la capacidad de computación y memoria de una computadora actual no permite disponer del espacio infinito de celdas, pero se soluciona de la siguiente manera: sea L un arreglo de celdas de longitud n la vecindad para la celda w_1 es $w_n w_1 w_2$ y la vecindad para la última celda w_n es $w_{n-1} w_n w_1$. De modo que los extremos del arreglo de celdas están conectados para dar continuidad espacial, esto soluciona muchos problemas sobre la limitación computacional.

Un autómata celular cuyas reglas generen un mecanismo capaz de simular el borrado y la adición de la cadena adjunta de cada regla de producción es posible. Así que definiremos al autómata celular unidimensional como:

$$A_C = (\Sigma_{AC}, L, u, f)$$

Donde:

- Σ_A es el alfabeto del autómata celular
- L es un arreglo de dimensión $d = 1$
- u es la vecindad que está conformada por una celda central y las celdas a sus laterales con un radio $r = 1$

- f es la función de transición que está conformada por las reglas de transición que se describen a continuación.

5.1.2. Reglas para el Borrado

En primer lugar, el sistema tag no tiene un número infinito de celdas donde desenvolverse, así que la extensión a los laterales de la cadena inicial se rellenará con un símbolo vacío o nulo N.

Luego es necesario simular el borrado del sistema tag, esta tarea se realizará mediante la adición de algunos símbolos de su alfabeto. Los símbolos adicionales y su aplicación son los siguientes:

- Para sustituir el primer carácter de la cadena y posteriormente saber que cadena se añadirá al final se necesita un número de símbolos auxiliares igual al número de símbolos que hay en el alfabeto del sistema tag. Como son dos $\{0, 1\}$ se utilizarán los símbolos auxiliares a, b cada uno va a corresponder al que sustituirán $a \to 0$ y $b \to 1$.

- Posteriormente, los símbolos auxiliares $\{a, b\}$ tienen que desplazarse al final de la cadena y mientras esto sucede hay que evitar que vuelvan a generarse en cada evolución si no hasta que termine de añadirse la cadena correspondiente en el extremo derecho, para lograr dicho bloqueo se utilizará el símbolo L. Este símbolo permanecerá en el extremo izquierdo de la cadena hasta que termine la concatenación de la cadena adjunta correspondiente.

- Para finalizar con el borrado debe haber un símbolo que sirva como bandera para el desbloqueo, que se elimine junto con el símbolo L y que pueda generarse otro de los símbolos auxiliares $\{a, b\}$ este símbolo será U. Así el conjunto de símbolos $\{L, U\}$ serán los símbolos de control o banderas.

Hasta el momento, para el borrado del sistema tag, el autómata celular usará el abecedario:

$$\Sigma_{AC} = \{N, 0, 1, a, b, L, U\}$$

Estas descripciones se tienen que expresar en reglas de transición. El símbolo X puede sustituirse con un símbolo cualquiera del abecedario del autómata celular.

N	0	X	→	a
N	1	X	→	b
X	N	a	→	L
X	N	b	→	L
X	L	0	→	L
X	L	1	→	L
L	U	0	→	N
L	U	1	→	N
X	L	U	→	N

Tabla 5.2: Reglas de transición para el borrado del sistema tag $0 \to 01, 1 \to 10;\ P = 1$

5.1.3. Reglas para el Desplazamiento de los Símbolos

Para determinar la cadena adjunta que se concatenará al final de la cada cadena según su primer carácter los símbolos auxiliares deben desplazarse a través de la cadena hasta el lado derecho de esta. Así mismo cuando se ha añadido la cadena adjunta es necesario desplazar la bandera U para desbloquear o eliminar la bandera L, todo esto entre los símbolos del alfabeto del sistema tag $\{0, 1\}$.

a	0	X	→	a
a	1	X	→	a
b	0	X	→	b
b	1	X	→	b
X	a	0	→	0
X	b	0	→	0
X	a	1	→	1
X	b	1	→	1

Tabla 5.3: Reglas de desplazamiento a la derecha para los símbolos auxiliares $\{a, b\}$.

L	0	U	→	U
0	0	U	→	U
1	0	U	→	U
L	1	U	→	U
0	1	U	→	U
1	1	U	→	U
0	U	X	→	0
1	U	X	→	1

Tabla 5.4: Reglas de desplazamiento a la izquierda para la bandera U.

5.1.4. Reglas para Concatenar la Cadena Adjunta

Como se ha visto hay un símbolo auxiliar por cada símbolo del alfabeto del sistema tag ($0 \rightarrow a$ y $1 \rightarrow b$) y se desplazarán desde el extremo izquierdo al extremo derecho de la cadena. Una vez que esto sucede se debe comenzar a concatenar la cadena adjunta, para lograr esto se necesitan otros símbolos auxiliares, el número de

estos símbolos auxiliares será dado por la longitud de cada cadena adjunta, es decir, en el caso de este sistema tag se tienen: $01, 10$. Por si solos, los símbolos a, b pueden añadir un carácter de la cadena adjunta que le corresponde a cada uno debido al alcance que tiene el autómata celular por el radio $r = 1$.

Por lo tanto, se deben añadir otros dos símbolos auxiliares $\{c, d\}$ porque cada cadena auxiliar es de longitud: 2. Así, cada uno de estos dos símbolos terminará de concatenar la cadena adjunta y generará la bandera de desbloqueo U. Tienen su correspondencia con los símbolos $\{a, b\}$ respectivamente ($a \rightarrow c$ y $b \rightarrow d$). Dicho funcionamiento se puede describir como:

- Cuando a llegue al final de la cadena de ceros y unos deberá realizar dos cosas: 1) agregar el símbolo 0 y 2) agregar el símbolo auxiliar c.

0	a	N	→	0
1	a	N	→	0
0	a	L	→	0
1	a	L	→	0
a	N	X	→	c
a	N	X	→	c
a	L	X	→	c
a	L	X	→	c

- Las funciones para el símbolo auxiliar c son: 1) agregar el símbolo 1 y 2) agregar la bandera U.

0	c	N	→	1
1	c	N	→	1
0	c	L	→	1
1	c	L	→	1
c	N	X	→	U
c	N	X	→	U
c	L	X	→	N
c	L	X	→	N

- Cuando b llegue al final de la cadena de ceros y unos deberá realizar dos cosas: 1) agregar el símbolo 1 y 2) agregar el símbolo auxiliar d.

0	b	N	→	1
1	b	N	→	1
0	b	L	→	1
1	b	L	→	1
b	N	X	→	d
b	N	X	→	d
b	L	X	→	d
b	L	X	→	d

- Las funciones para el símbolo auxiliar d son: 1) agregar el símbolo 0 y 2) agregar la bandera U.

0	d	N	→	0
1	d	N	→	0
0	d	L	→	0
1	d	L	→	0
d	N	X	→	U
d	N	X	→	U
d	L	X	→	N
d	L	X	→	N

De esta manera se habrán agregado otros dos símbolos auxiliares al alfabeto del autómata celular. Estos son los últimos símbolos así que el alfabeto completo es el siguiente:

$$\Sigma_{AC} = \{N, 0, 1, a, b, c, d, L, U\}$$

5.1.5. Reglas de Estabilidad

Mientras cada una de las reglas anteriormente definidas dan lugar al mecanismo del sistema tag es necesario identificar las reglas de transición capaces de estabilizar el funcionamiento del autómata celular. Como se mencionó existen símbolos de control o banderas que permiten controlar la generación de los símbolos auxiliares, pero hay que permitir que existe estabilidad en cada evolución. Es decir, para los símbolos $\{N, 1, 0\}$ se deben establecer reglas que especifiquen un espacio estable. Estas reglas son:

N	N	N	→	N
N	N	L	→	N
N	N	0	→	N
N	N	1	→	N
0	N	N	→	N
0	N	L	→	N
1	N	N	→	N
1	N	L	→	N
U	N	N	→	N
U	N	L	→	N

Tabla 5.5: Reglas de estabilidad para el espacio vacío.

0	0	0	→	0
0	0	1	→	0
0	0	N	→	0
0	0	L	→	0
0	0	a	→	0
0	0	b	→	0
0	0	c	→	0
0	0	d	→	0
1	0	0	→	0
1	0	1	→	0
1	0	N	→	0
1	0	L	→	0

1	0	a	→	0
1	0	b	→	0
1	0	c	→	0
1	0	d	→	0
L	0	0	→	0
L	0	1	→	0
L	0	N	→	0
L	0	L	→	0
L	0	a	→	0
L	0	b	→	0
L	0	c	→	0
L	0	d	→	0
U	0	0	→	0
U	0	1	→	0
U	0	N	→	0
U	0	L	→	0
U	0	a	→	0
U	0	b	→	0
U	0	c	→	0
U	0	d	→	0
0	1	0	→	1
0	1	1	→	1
0	1	N	→	1
0	1	L	→	1
0	1	a	→	1
0	1	b	→	1
0	1	c	→	1
0	1	d	→	1

1	1	0	→	1
1	1	1	→	1
1	1	N	→	1
1	1	L	→	1
1	1	a	→	1
1	1	b	→	1
1	1	c	→	1
1	1	d	→	1
L	1	0	→	1
L	1	1	→	1
L	1	N	→	1
L	1	L	→	1
L	1	a	→	1
L	1	b	→	1
L	1	c	→	1
L	1	d	→	1
U	1	0	→	1
U	1	1	→	1
U	1	N	→	1
U	1	L	→	1
U	1	a	→	1
U	1	b	→	1
U	1	c	→	1
U	1	d	→	1

Tabla 5.6: Reglas de estabilidad para los unos y ceros.

En este autómata celular y en los posteriores se establece una regla general pa-

ra cualquier otro caso en el que no exista una regla para la vecindad $w_{n-1}w_nw_{n+1}$, $w_{n-1}, w_n, w_{n+1} \in \Sigma_{AC}$.

w_{n-1}	w_n	w_{n+1}	$\rightarrow$	w_{n-1}

Tabla 5.7: Regla para cualquier otro caso.

5.1.6. Pruebas de Simulación del Autómata Celular Unidimensional

Una vez modelado el autómata celular unidimensional que simulará el sistema tag de la sucesión de Thue-Morse se obtienen las evoluciones para comprobar que el modelo se estableció correctamente. De esta manera se obtienen las primeras 51 evoluciones del autómata con una cadena inicial $N0NNNNNNNNNN$ que es equivalente a la cadena 0 en el sistema tag. Posteriormente, se utilizarán colores para observar la dinámica de manera sencilla.

En la siguiente tabla se tienen los valores de cada celda donde la longitud del espacio de las celdas es de 14. En las primeras 51 evoluciones de este autómata se obtienen 6 de las cadenas resultantes en el sistema tag de la sucesión de Thue-Morse. De manera gráfica está en la Figura 5.2 y en la Figura 5.3 se presentan 500 evoluciones con una configuración inicial aleatoria.

N	0	N	N	N	N	N	N	N	N	N	N	N
N	a	N	N	N	N	N	N	N	N	N	N	N
L	0	c	N	N	N	N	N	N	N	N	N	N
L	0	1	U	N	N	N	N	N	N	N	N	N
L	0	U	1	N	N	N	N	N	N	N	N	N
L	U	0	1	N	N	N	N	N	N	N	N	N
N	N	0	1	N	N	N	N	N	N	N	N	N
N	N	a	1	N	N	N	N	N	N	N	N	N
N	L	1	a	N	N	N	N	N	N	N	N	N
N	L	1	0	c	N	N	N	N	N	N	N	N
N	L	1	0	1	U	N	N	N	N	N	N	N
N	L	1	0	U	1	N	N	N	N	N	N	N
N	L	1	U	0	1	N	N	N	N	N	N	N
N	L	U	1	0	1	N	N	N	N	N	N	N
N	N	N	1	0	1	N	N	N	N	N	N	N
N	N	N	b	0	1	N	N	N	N	N	N	N
N	N	L	0	b	1	N	N	N	N	N	N	N
N	N	L	0	1	b	N	N	N	N	N	N	N
N	N	L	0	1	1	d	N	N	N	N	N	N
N	N	L	0	1	1	0	U	N	N	N	N	N
N	N	L	0	1	1	U	0	N	N	N	N	N
N	N	L	0	1	U	1	0	N	N	N	N	N
N	N	L	0	U	1	1	0	N	N	N	N	N
N	N	L	U	0	1	1	0	N	N	N	N	N
N	N	N	N	0	1	1	0	N	N	N	N	N
N	N	N	N	a	1	1	0	N	N	N	N	N
N	N	N	L	1	a	1	0	N	N	N	N	N

N	N	N	L	1	1	a	0	N	N	N	N	N
N	N	N	L	1	1	0	a	N	N	N	N	N
N	N	N	L	1	1	0	0	c	N	N	N	N
N	N	N	L	1	1	0	0	1	U	N	N	N
N	N	N	L	1	1	0	0	U	1	N	N	N
N	N	N	L	1	1	0	U	0	1	N	N	N
N	N	N	L	1	1	U	0	0	1	N	N	N
N	N	N	L	1	U	1	0	0	1	N	N	N
N	N	N	L	U	1	1	0	0	1	N	N	N
N	N	N	N	N	1	1	0	0	1	N	N	N
N	N	N	N	N	b	1	0	0	1	N	N	N
N	N	N	N	L	1	b	0	0	1	N	N	N
N	N	N	N	L	1	0	b	0	1	N	N	N
N	N	N	N	L	1	0	0	b	1	N	N	N
N	N	N	N	L	1	0	0	1	b	N	N	N
N	N	N	N	L	1	0	0	1	1	d	N	N
N	N	N	N	L	1	0	0	1	1	0	U	N
N	N	N	N	L	1	0	0	1	1	U	0	N
N	N	N	N	L	1	0	0	1	U	1	0	N
N	N	N	N	L	1	0	0	U	1	1	0	N
N	N	N	N	L	1	0	U	0	1	1	0	N
N	N	N	N	L	1	U	0	0	1	1	0	N
N	N	N	N	L	U	1	0	0	1	1	0	N
N	N	N	N	N	N	1	0	0	1	1	0	N

Tabla 5.8: Primeras 51 evoluciones del autómata celular que simula el sistema tag de la sucesión de Thue-Morse con la cadena inicial $N0NNNNNNNNNN$.

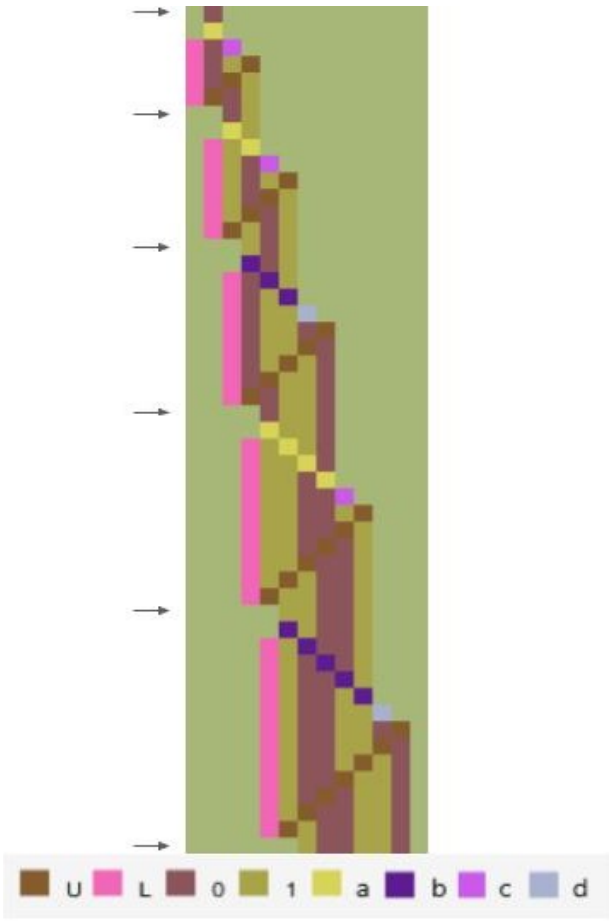


Figura 5.2: Primeras 51 evoluciones del autómata celular que simula el sistema tag de la sucesión de Thue-Morse con la cadena inicial $N0NNNNNNNNNNN$.

Figura 5.3: Primeras 500 evoluciones del autómata celular que simula el sistema tag de la sucesión de Thue-Morse con una cadena inicial aleatoria.

5.2. Sistema Tag del Problema de la Etiqueta

Como se vio en el capítulo 3 el sistema tag del problema de la etiqueta ha sido objeto de estudio para investigadores como Emil Post [8], Marvin Minsky [5] y Stephen Wolfram [12]. Emil Post lo catalogó como un problema intratable, Marvin Minsky encontró que en general los sistemas tag son indecidibles y Stephen Wolfram, a pesar de intentar reducirlo, lo ha encontrado irreductible aún.

En esta ocasión se propone un autómata celular unidimensional capaz de simular el famoso sistema tag:

$$0 \to 00, 1 \to 1101$$

$$P = 3$$

De antemano, en la tabla 5.9 y la figura 5.4 se observan las primeras transiciones de dicho sistema tag.

1	0	0	1	0	0	1	0	0	1						
1	0	0	1	0	0	1	1	1	0	1					
1	0	0	1	1	1	0	1	1	1	0	1				
1	1	1	0	1	1	1	0	1	1	1	0	1			
0	1	1	1	0	1	1	1	0	1	1	1	0	1		
1	0	1	1	1	0	1	1	1	0	1	0	0			
1	1	0	1	1	1	0	1	0	0	1	1	0	1		
1	1	1	0	1	0	0	1	1	0	1	1	1	0	1	
0	1	0	0	1	1	0	1	1	1	0	1	1	1	0	1
0	1	1	0	1	1	1	0	1	1	1	0	1	0	0	
0	1	1	1	0	1	1	1	0	1	0	0	0	0		
1	0	1	1	1	0	1	0	0	0	0	0	0			
1	1	0	1	0	0	0	0	0	0	1	1	0	1		
1	0	0	0	0	0	0	1	1	0	1	1	1	0	1	
0	0	0	0	1	1	0	1	1	1	0	1	1	1	0	1
0	1	1	0	1	1	1	0	1	1	1	0	1	0	0	
0	1	1	1	0	1	1	1	0	1	0	0	0	0		
1	0	1	1	1	0	1	0	0	0	0	0	0			

Tabla 5.9: Primeras 18 transiciones del sistema tag $0 \to 00, 1 \to 1101$; $P = 3$ con la cadena inicial 1001001001.

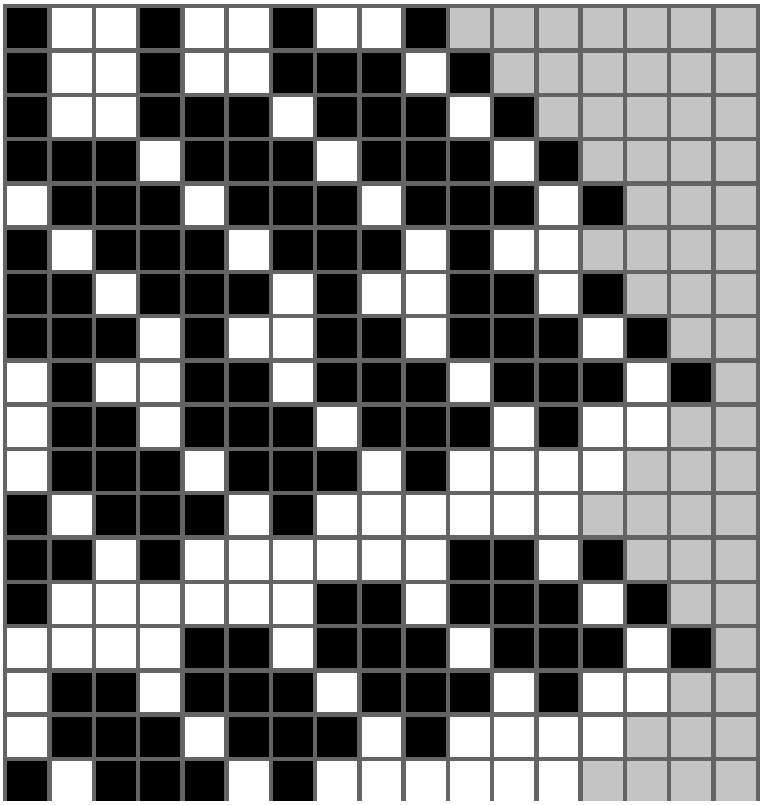


Figura 5.4: Primeras 18 transiciones del sistema tag $0 \rightarrow 00, 1 \rightarrow 1101$; $P = 3$ con la cadena inicial 1001001001.

5.2.1. Método Heurístico para encontrar el Autómata Celular Equivalente

Un autómata celular cuyas reglas generen un mecanismo capaz de simular el borrado y la adición de la cadena adjunta de cada regla de producción es posible. Así que definiremos al autómata celular unidimensional como:

$$A_C = (\Sigma_{AC}, L, u, f)$$

Donde:

- Σ_{AC} es el alfabeto del autómata celular

- L es un arreglo de dimensión $d = 1$
- u es la vecindad que está conformada por una celda central y las celdas a sus laterales con un radio $r = 1$
- f es la función de transición que está conformada por las reglas de transición que se describen a continuación.

Al utilizar los datos que se pueden obtener de la definición de este sistema tag tales como:

- El abecedario como $\Sigma_{TS} = \{0, 1\}$.
- El conjunto de reglas de producción como $R_{TS} = \{0 \to 00, 1 \to 1101\}$.
- El conjunto de cadenas adjuntas como $A_{TS} = \{00, 1101\}$.
- El número de borrado $P = 3$.

El alfabeto Σ_{AC} se obtiene por:

$$\Sigma_{AC} = \{N\} \cup \Sigma_{CT} \cup \Sigma_{TS} \cup \Sigma_P \cup \Sigma_S \cup \Sigma_A$$

Donde:

- $\{N\}$ es el símbolo del espacio nulo.
- $\Sigma_{CT} = \{L, U\}$ el conjunto de símbolos de control o banderas.
- $\Sigma_{TS} = \{0, 1\}$ el alfabeto del sistema tag.
- Σ_P es el conjunto de símbolos para borrado.
- Σ_S es el conjunto de símbolos para envío de señal.
- Σ_A es el conjunto de símbolos para adjuntar cadenas.

5.2.2. Borrado y Bloqueo

En la subsección 4.1.2 se definió un mecanismo y reglas para hacer el borrado de un sistema tag con $P = 1$ y gracias a esa característica fue posible el funcionamiento, pero para un sistema tag con $P = 3$ no es posible aplicar al menos en parte este funcionamiento, pero existe una compatibilidad entre el modelo de este autómata celular y el anterior.

Como se hizo para el autómata celular de la sección anterior el alfabeto del sistema tag $\Sigma_{TS} = \{0, 1\}$ se necesita un símbolo auxiliar por cada uno perteneciente al alfabeto Σ_S para representar un envío de señal que se detallará en la siguiente subsección. Para esta sección el alfabeto Σ_S se utilizará también en el borrado, y se obtiene de:

$$|\Sigma_S| = |\Sigma_{TS}| = 2$$

De manera arbitraria se utilizarán:

$$\Sigma_S = \{a, b\}$$

Como es necesario saber cuál es el primer símbolo de la cadena y enviar una señal diferente para cada $\sigma_i \in \Sigma_{TS}$, se tienen que utilizar símbolos diferentes para el borrado en cada caso contando los de Σ_S. Σ_P es el alfabeto que tiene exclusivamente símbolos auxiliares de borrado. Los cuales se obtienen de:

$$|\Sigma_P| = |\Sigma_{TS}|(P - 1) = 2(3 - 1) = 2(2) = 4$$

De manera arbitraria se utilizarán:

$$\Sigma_P = \{w, x, y, z\}$$

Es importante sustituir los primeros tres caracteres de la cadena por el símbolo N para simular el borrado $P = 3$ pero el autómata celular debe poder diferenciar cuál era el primer símbolo para saber que cadena concatenar al final, $\{a, b\}$ como ya vimos, ayudan a borrar un carácter y para borrar los otros dos se usaran los símbolos de Σ_P, dos para cada caso. Si el primer símbolo es 0 entonces los símbolos auxiliares del borrado serán $\{w, y\}$ y si el primer símbolo es 1 los símbolos auxiliares del borrado serán $\{x, z\}$. De manera que cuando el primer carácter de la cadena sea 0 se sustituirá por w y el siguiente carácter se sustituirá por y y finalmente el siguiente por a gracias a que como en el autómata anterior el símbolo auxiliar a corresponde al 0 para que se desplace al final de la cadena y saber que cadena adjunta se concatenará. Las sucesiones entre cada símbolo auxiliar de borrado se aplicarán según el primer carácter de la cadena, cada dos $p_i \in \Sigma_P$ pertenecerán al borrado de cada $\sigma_i \in \Sigma_{TS}$.

$$0 \to w \to y \to a \to L$$

$$1 \to x \to z \to b \to L$$

Para finalizar el borrado sucederá exactamente de la misma manera que el autómata celular de la sección 4.1 en el cual se eliminan L y U cuando están a un lado del otro.

Estas descripciones se tienen que expresar en reglas de transición, simplificándolas con expresiones regulares. El símbolo X puede sustituirse con un símbolo cualquiera del abecedario del autómata celular.

N	0	Σ^+_{AC}	$\rightarrow$	w
N	1	Σ^+_{AC}	$\rightarrow$	x
w	Σ^+_{AC}	Σ^+_{AC}	$\rightarrow$	y
x	Σ^+_{AC}	Σ^+_{AC}	$\rightarrow$	z
y	Σ^+_{AC}	Σ^+_{AC}	$\rightarrow$	a
z	Σ^+_{AC}	Σ^+_{AC}	$\rightarrow$	b
N	(w + x + y + z)	Σ^+_{AC}	$\rightarrow$	N
Σ^+_{AC}	N	(a + b)	$\rightarrow$	L
Σ^+_{AC}	L	(0 + 1)	$\rightarrow$	L
L	U	(0 + 1)	$\rightarrow$	N
X	L	U	$\rightarrow$	N

Tabla 5.10: Reglas de transición para el borrado del sistema tag $0 \rightarrow 00, 1 \rightarrow 1101;\ P = 3$.

5.2.3. Envío de Señal

El alfabeto Σ_S definido para contener los símbolos auxiliares que se encargarán de ser la señal que llegará del principio de la cadena hasta el final para saber que cadena adjunta se concatenará. Cada $s_i \in \Sigma_S$ corresponde a un $\sigma_i \in \Sigma_{TS}$.

$$0 \rightarrow a$$

$$1 \rightarrow b$$

La compatibilidad de esta definición de autómata celular con el anterior permite que las mismas reglas de desplazamiento del autómata celular anterior puedan aplicarse a este, finalmente se tienen que desplazar los símbolos $\{a, b\}$ a la derecha. En esta ocasión se muestran las reglas simplificadas por expresiones regulares.

a	$(0+1)$	Σ^{+}_{AC}	$\rightarrow$	a
b	$(0+1)$	Σ^{+}_{AC}	$\rightarrow$	b
Σ^{+}_{AC}	(a + b)	0	$\rightarrow$	0
Σ^{+}_{AC}	(a + b)	1	$\rightarrow$	1

Tabla 5.11: Reglas de transición para el envío de señal del sistema tag $0 \rightarrow 00, 1 \rightarrow 1101;\ P = 3$.

5.2.4. Reglas para Concatenar la Cadena Adjunta

El último alfabeto Σ_A que es el que contendrá los símbolos auxiliares para concatenar la cadena adjunta. El número de símbolos a utilizar se calcula con la suma de cada $|A_i| > 0$ donde $A_i \in A_{TS}$.

$$|\Sigma_A| = \sum_{i=1}^{|A_{TS}|} |A_i| - 1, |A_i| > 0$$

$$|\Sigma_A| = 4$$

Gracias a que se tienen los símbolos auxiliares $\{a, b\}$ que corresponden con $0 \rightarrow a$ y $1 \rightarrow b$ y estos dos a su vez pueden añadir un símbolo de su respectiva A_i. Es decir, que para la cadena adjunta $A_1 = 00$ se necesita 1 símbolo auxiliar y para $A_2 = 1101$ son 3 símbolos auxiliares, estos son: $\{c\}$ y $\{d, e, f\}$ respectivamente.

$$00 \Rightarrow a \rightarrow c \rightarrow U$$

$$1101 \Rightarrow b \rightarrow d \rightarrow e \rightarrow f \rightarrow U$$

Cuyas reglas con expresiones regulares son las siguientes:

a	(N + L)	Σ^{+}_{AC}	→	c
(N + 0 + 1)	a	(N + L)	→	0
c	(N + L)	Σ^{+}_{AC}	→	U
(N + 0 + 1)	c	(N + L)	→	0

Tabla 5.12: Reglas para la concatenación de la cadena 00.

b	(N + L)	Σ^{+}_{AC}	→	d
(N + 0 + 1)	b	(N + L)	→	1
d	(N + L)	Σ^{+}_{AC}	→	e
(N + 0 + 1)	d	(N + L)	→	1
e	(N + L)	Σ^{+}_{AC}	→	f
(N + 0 + 1)	e	(N + L)	→	0
f	(N + L)	Σ^{+}_{AC}	→	U
(N + 0 + 1)	f	(N + L)	→	1

Tabla 5.13: Reglas para la concatenación de la cadena 1101.

Las reglas anteriormente descritas en realidad funcionan de la misma manera para cada caso y a partir de este punto se obtiene el alfabeto del autómata celular A_C.

$$\Sigma_{CA} = \{N, 0, 1, w, x, y, z, a, b, c, d, e, f, L, U\}$$

5.2.5. Desbloqueo

Cuando la bandera U está al final de la cadena se desplazará al principio de esta para quitar la bandera L para comenzar el proceso de nuevo. Este comportamiento se expresa en las siguientes reglas:

0	U	Σ^{+}_{AC}	$\rightarrow$	0
1	U	Σ^{+}_{AC}	$\rightarrow$	1
(L + 0 + 1)	(0 + 1)	U	$\rightarrow$	U

Tabla 5.14: Reglas de para el desbloqueo.

5.2.6. Reglas de Estabilidad

Mientras cada una de las reglas anteriormente definidas dan lugar al mecanismo del sistema tag es necesario identificar las reglas de transición capaces de estabilizar el funcionamiento del autómata celular. Es decir, para los símbolos $\{N, 1, 0\}$ se deben establecer reglas que especifiquen un espacio estable. Estas reglas son:

N	N	(N + L + 0 + 1 + w + x + y + z)	$\rightarrow$	N
(U + 0 + 1 + w + x + y + z)	N	(N + L)	$\rightarrow$	N

Tabla 5.15: Reglas de estabilidad para el espacio vacío.

(L + U + 0 + 1)	0	(N + L + 0 + 1 + a + b + c + d + e + f)	$\rightarrow$	0
(L + U + 0 + 1)	1	(N + L + 0 + 1 + a + b + c + d + e + f)	$\rightarrow$	1

Tabla 5.16: Reglas de estabilidad para los ceros y unos.

5.2.7. Pruebas de Simulación del Autómata Celular Unidimensional

Para comprobar que el modelo de autómata celular unidimensional realmente simula el Sistema Tag del Problema de la Etiqueta es necesario hacer una prueba de escritorio. De esta manera se obtienen las primeras 47 evoluciones del autómata

con una cadena inicial $N100011100NNNNNNNNNN$ que es equivalente a la cadena 100011100 en el sistema tag. Posteriormente, se utilizarán colores para observar la dinámica de manera sencilla.

En la siguiente tabla se tienen los valores de cada celda donde la longitud del espacio de las celdas es de 20. En las primeras 47 evoluciones de este autómata se obtienen 3 de las cadenas resultantes en el sistema tag. De manera gráfica está en la Figura 5.5 y en la Figura 5.6 se encuentran las primeras 500 evoluciones de una cadena inicial aleatoria.

N	1	0	0	0	1	1	1	0	0	N	N	N	N	N	N	N	N	N	N
N	x	0	0	0	1	1	1	0	0	N	N	N	N	N	N	N	N	N	N
N	N	z	0	0	1	1	1	0	0	N	N	N	N	N	N	N	N	N	N
N	N	N	b	0	1	1	1	0	0	N	N	N	N	N	N	N	N	N	N
N	N	L	0	b	1	1	1	0	0	N	N	N	N	N	N	N	N	N	N
N	N	L	0	1	b	1	1	0	0	N	N	N	N	N	N	N	N	N	N
N	N	L	0	1	1	b	1	0	0	N	N	N	N	N	N	N	N	N	N
N	N	L	0	1	1	1	b	0	0	N	N	N	N	N	N	N	N	N	N
N	N	L	0	1	1	1	0	b	0	N	N	N	N	N	N	N	N	N	N
N	N	L	0	1	1	1	0	0	b	N	N	N	N	N	N	N	N	N	N
N	N	L	0	1	1	1	0	0	1	d	N	N	N	N	N	N	N	N	N
N	N	L	0	1	1	1	0	0	1	1	e	N	N	N	N	N	N	N	N
N	N	L	0	1	1	1	0	0	1	1	0	f	N	N	N	N	N	N	N
N	N	L	0	1	1	1	0	0	1	1	0	1	U	N	N	N	N	N	N
N	N	L	0	1	1	1	0	0	1	1	0	U	1	N	N	N	N	N	N
N	N	L	0	1	1	1	0	0	1	1	U	0	1	N	N	N	N	N	N
N	N	L	0	1	1	1	0	0	1	U	1	0	1	N	N	N	N	N	N

N	N	L	0	1	1	1	0	0	U	1	1	0	1	N	N	N	N	N	N
N	N	L	0	1	1	1	0	U	0	1	1	0	1	N	N	N	N	N	N
N	N	L	0	1	1	1	U	0	0	1	1	0	1	N	N	N	N	N	N
N	N	L	0	1	1	U	1	0	0	1	1	0	1	N	N	N	N	N	N
N	N	L	0	1	U	1	1	0	0	1	1	0	1	N	N	N	N	N	N
N	N	L	0	U	1	1	1	0	0	1	1	0	1	N	N	N	N	N	N
N	N	L	U	0	1	1	1	0	0	1	1	0	1	N	N	N	N	N	N
N	N	N	N	0	1	1	1	0	0	1	1	0	1	N	N	N	N	N	N
N	N	N	N	w	1	1	1	0	0	1	1	0	1	N	N	N	N	N	N
N	N	N	N	N	y	1	1	0	0	1	1	0	1	N	N	N	N	N	N
N	N	N	N	N	N	a	1	0	0	1	1	0	1	N	N	N	N	N	N
N	N	N	N	N	L	1	a	0	0	1	1	0	1	N	N	N	N	N	N
N	N	N	N	N	L	1	0	a	0	1	1	0	1	N	N	N	N	N	N
N	N	N	N	N	L	1	0	0	a	1	1	0	1	N	N	N	N	N	N
N	N	N	N	N	L	1	0	0	1	a	1	0	1	N	N	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	a	0	1	N	N	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	0	a	1	N	N	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	0	1	a	N	N	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	0	1	0	c	N	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	0	1	0	0	U	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	0	1	0	U	0	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	0	1	U	0	0	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	0	U	1	0	0	N	N	N	N
N	N	N	N	N	L	1	0	0	1	1	U	0	1	0	0	N	N	N	N
N	N	N	N	N	L	1	0	0	1	U	1	0	1	0	0	N	N	N	N
N	N	N	N	N	L	1	0	0	U	1	1	0	1	0	0	N	N	N	N

N	N	N	N	N	L	1	0	U	0	1	1	0	1	0	0	N	N	N	N
N	N	N	N	N	L	1	U	0	0	1	1	0	1	0	0	N	N	N	N
N	N	N	N	N	L	U	1	0	0	1	1	0	1	0	0	N	N	N	N
N	N	N	N	N	N	N	1	0	0	1	1	0	1	0	0	N	N	N	N

Tabla 5.17: Primeras 47 evoluciones del autómata celular que simula el sistema tag del Problema de la Etiqueta con la cadena inicial $N100011100NNNNNNNNN$.

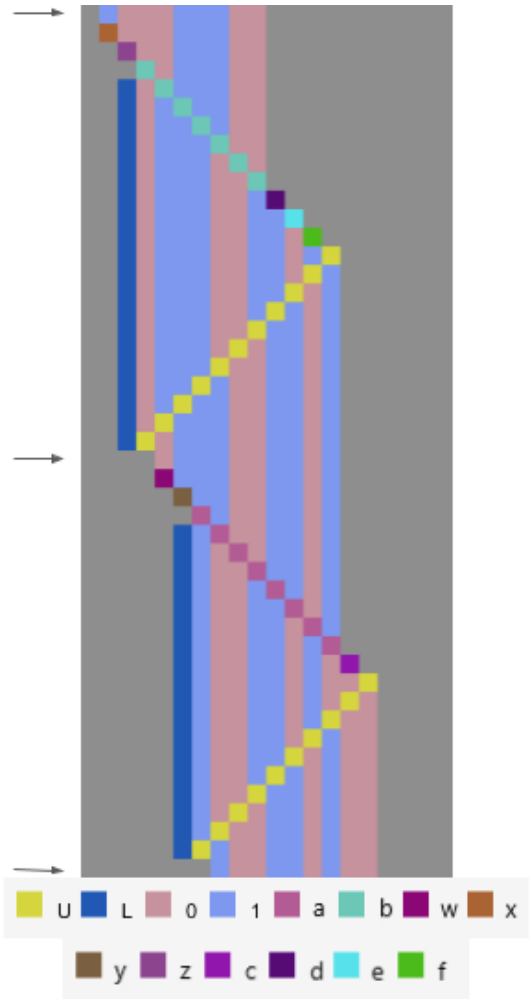


Figura 5.5: Primeras 47 evoluciones del autómata celular que simula el sistema tag del Problema de la Etiqueta con la cadena inicial $N100011100NNNNNNNNN$.

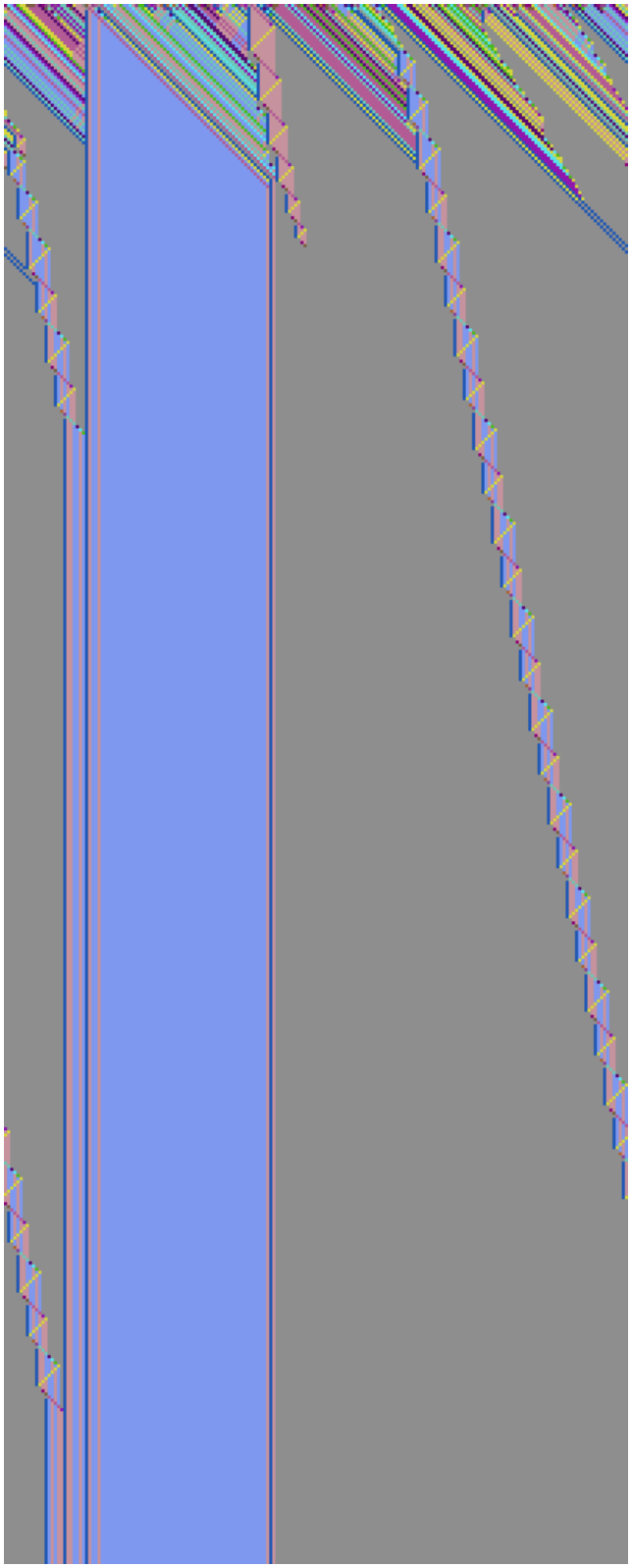


Figura 5.6: Primeras 500 evoluciones del autómata celular que simula el sistema tag del Problema de la Etiqueta con una cadena inicial aleatoria.

5.3. Algoritmo basado en el método heurístico

El establecimiento de un algoritmo capaz de calcular el autómata celular equivalente a un sistema tag arbitrario se considera de suma importancia para explorar la dinámica espacial de los sistemas tag en autómatas celulares y proporciona muchas posibilidades en relación con los sistemas tag como sistemas dinámicos discretos como dotarlos de la capacidad de computación paralela.

Por lo tanto, basado en el método heurístico de la sección 5.2 para el sistema tag del problema de la etiqueta, se puede determinar un algoritmo que, dado un sistema tag cualquiera, resulte en un autómata celular equivalente.

Sean $T = (\Sigma_{TS}, H, R_{TS}, P)$ un sistema tag y $A_C = (\Sigma_{AC}, L, u, f)$ un autómata celular unidimensional cuya vecindad u será de un radio $r = 1$. El algoritmo se expresa a continuación:

$\Sigma_{CT} = \{ L, U \}, \Sigma_P = \emptyset, \Sigma_S = \emptyset, \Sigma_A = \emptyset$
Añadir $|\Sigma_{TS}|(P-1)$ símbolos auxiliares a Σ_P
Añadir $|\Sigma_{TS}|$ símbolos auxiliares a Σ_S
$i = 1$
Para cada A_i en R_{TS}:
    Si $|A_i| > 0$ entonces, añadir $|A_i| - 1$ símbolos auxiliares a Σ_A
    $i++$
$\Sigma_{AC} = \{N\} \cup \Sigma_{TS} \cup \Sigma_{CT} \cup \Sigma_S \cup \Sigma_P \cup \Sigma_A$
Si $|\Sigma_P| > 0$ entonces, añadir $N({\Sigma_P}^+)({\Sigma_{AC}}^+) \to N$ a f
Añadir $({\Sigma_{AC}}^+)N({\Sigma_S}^+) \to L$ a f
Añadir $({\Sigma_{AC}}^+)L({\Sigma_{TS}}^+) \to L$ a f
Añadir $NN(N + L + {\Sigma_{TS}}^+ + {\Sigma_P}^+) \to N$ a f
Añadir $(U + {\Sigma_{TS}}^+ + {\Sigma_{TS}}^P)N(N + L) \to N$ a f
Añadir $(L + {\Sigma_{TS}}^+)({\Sigma_{TS}}^+)U \to U$ a f
Añadir $LU({\Sigma_{TS}}^+) \to N$ a f
Añadir $({\Sigma_{CA}}^+)LU \to N$ a f
$i = 1$
Para cada σ_i, s_i, p_i en $\Sigma_{TS}, \Sigma_S, \Sigma_P$:
    Añadir $(L + U + {\Sigma_{TS}}^+)\sigma_i(N + L + {\Sigma_{TS}}^+ + {\Sigma_S}^+ + {\Sigma_A}^+) \to \sigma_i$ a f
    Añadir $({\Sigma_{AC}}^+)({\Sigma_S}^+)\sigma_i \to \sigma_i$ a f
    Añadir $\sigma_i U({\Sigma_{AC}}^+) \to \sigma_i$ a f
    Añadir $s_i({\Sigma_{TS}}^+)({\Sigma_{AC}}^+) \to s_i$ a f
    Si $|\Sigma_P| > 0$ entonces, añadir $N\sigma_i({\Sigma_{AC}}^+) \to p_i$ a f
    Si no, añadir $N\sigma_i({\Sigma_{AC}}^+) \to s_i$ a f
    $i++$
$i = 1$
$\Sigma_{PS} = \Sigma_P \cup \Sigma_S$
Para cada p_i, ps_i en Σ_P, Σ_{PS}:
    Añadir $ps_i({\Sigma_{AC}}^+)({\Sigma_{AC}}^+) \to ps_{i+1}$ a f
    $i++$
$i = 1$
$k = 1$
Para cada A_i, s_i en R_{TS}, Σ_S:
    $j = 1$
    Si $|A_i| > 0$ entonces,
        Para cada ω_j, a_k en A_i, Σ_A:
            Si $j = 1$ entonces,
                Si $j = |A_i|$ entonces, añadir $s_i(N + L)\left({\Sigma_{AC}}^+\right) \to U$ a f
                Si no, añadir $s_i(N + L)({\Sigma_{AC}}^+) \to a_k$ a f
                Añadir $(\mathrm{N} + {\Sigma_{TS}}^+)s_i(N + L) \to \omega_j$ a f
            Si no,
                Si $j = |A_i|$ entonces, añadir $a_k(N + L)({\Sigma_{AC}}^+) \to U$ a f
                Si no, añadir $a_k(N + L)({\Sigma_{AC}}^+) \to a_{k+1}$ a f
                Añadir $(\mathrm{N} + {\Sigma_{TS}}^+)a_k(N + L) \to \omega_j$ a f
                $k++$
            $j++$
    Si no,
        Añadir $(N + {\Sigma_{TS}}^+)s_i({\Sigma_{AC}}^+) \to U$ a f
        Añadir $s_i(N + L)({\Sigma_{AC}}^+) \to N$ a f
    $i++$

Figura 5.7: Algoritmo que, dado un sistema tag cualquiera, resulta en un autómata celular equivalente.

5.3.1. Ejemplo de aplicación del algoritmo

Para el sistema tag de la función de Collatz de la sección 4.6 se tiene

$$T = (\Sigma_{TS} = \{0, 1, 2\}, H = \emptyset, R_{TS} = \{0 \to 12, 1 \to 0, 2 \to 000\}, P = 2)$$

que resulta en un autómata celular $A_C = (\Sigma_{AC}, L, u, f)$ donde,

- $\Sigma_{AC} = \{N, U, L, 0, 1, 2, A, B, C, D, E, F, G, H, I\}$
- L es un arreglo de dimensión $d = 1$
- u es la vecindad que está conformada por una celda central y las celdas a sus laterales con un radio $r = 1$
- f es la función de transición que está conformada por las reglas de transición que se describen a continuación.

N	N	(N+L+0+1+2+D+E+F)	→	N
(U+0+1+2+D+E+F)	N	(N+L)	→	N
N	(D+E+F)	Σ^{+}_{AC}	→	N
Σ^{+}_{AC}	N	(A+B+C)	→	L
Σ^{+}_{AC}	L	(0+1+2)	→	L
D	Σ^{+}_{AC}	Σ^{+}_{AC}	→	A
E	Σ^{+}_{AC}	Σ^{+}_{AC}	→	B
F	Σ^{+}_{AC}	Σ^{+}_{AC}	→	C
A	(0+1+2)	Σ^{+}_{AC}	→	A
B	(0+1+2)	Σ^{+}_{AC}	→	B
C	(0+1+2)	Σ^{+}_{AC}	→	C
Σ^{+}_{AC}	(A+B+C)	0	→	0
Σ^{+}_{AC}	(A+B+C)	1	→	1
Σ^{+}_{AC}	(A+B+C)	2	→	2
N	0	Σ^{+}_{AC}	→	D
N	1	Σ^{+}_{AC}	→	E
N	2	Σ^{+}_{AC}	→	F
0	U	Σ^{+}_{AC}	→	0
1	U	Σ^{+}_{AC}	→	1
2	U	Σ^{+}_{AC}	→	2
L	U	(0+1+2)	→	N
Σ^{+}_{AC}	L	U	→	N

(L+U+0+1+2)	0	(N+L+0+1+2+A+B+C+G+H+I)	→	0
(L+U+0+1+2)	1	(N+L+0+1+2+A+B+C+G+H+I)	→	2
(L+U+0+1+2)	1	(N+L+0+1+2+A+B+C+G+H+I)	→	2
A	(N+L)	Σ^{+}_{AC}	→	G
(N+0+1+2)	A	(N+L)	→	1
G	(N+L)	Σ^{+}_{AC}	→	U
(N+0+1+2)	G	(N+L)	→	2
B	(N+L)	Σ^{+}_{AC}	→	U
(N+0+1+2)	B	(N+L)	→	0
C	(N+L)	Σ^{+}_{AC}	→	H
(N+0+1+2)	C	(N+L)	→	0
H	(N+L)	Σ^{+}_{AC}	→	I
(N+0+1+2)	H	(N+L)	→	0
I	(N+L)	Σ^{+}_{AC}	→	U
(N+0+1+2)	I	(N+L)	→	0
(L+0+1+2)	(0+1+2)	U	→	U

Tabla 5.18: Reglas de transición del autómata celular equivalente al sistema tag $0 \to 12, 1 \to 0, 2 \to 000;\ P = 2$.

Del cual se muestran las siguientes simulaciones: la primera con la cadena inicial 00000 y la segunda con una cadena inicial aleatoria.

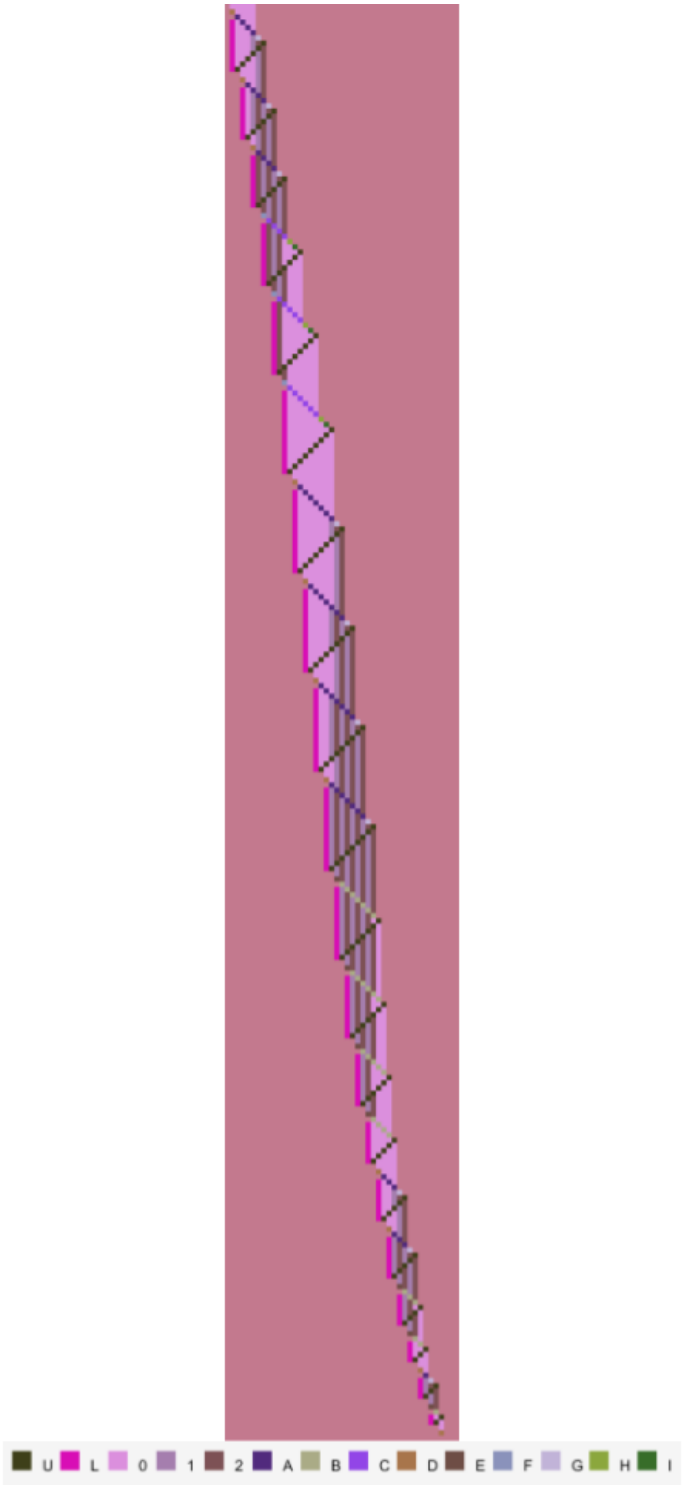


Figura 5.8: Primeras 275 evoluciones con la cadena inicial 00000 en un arreglo de 45 celdas.

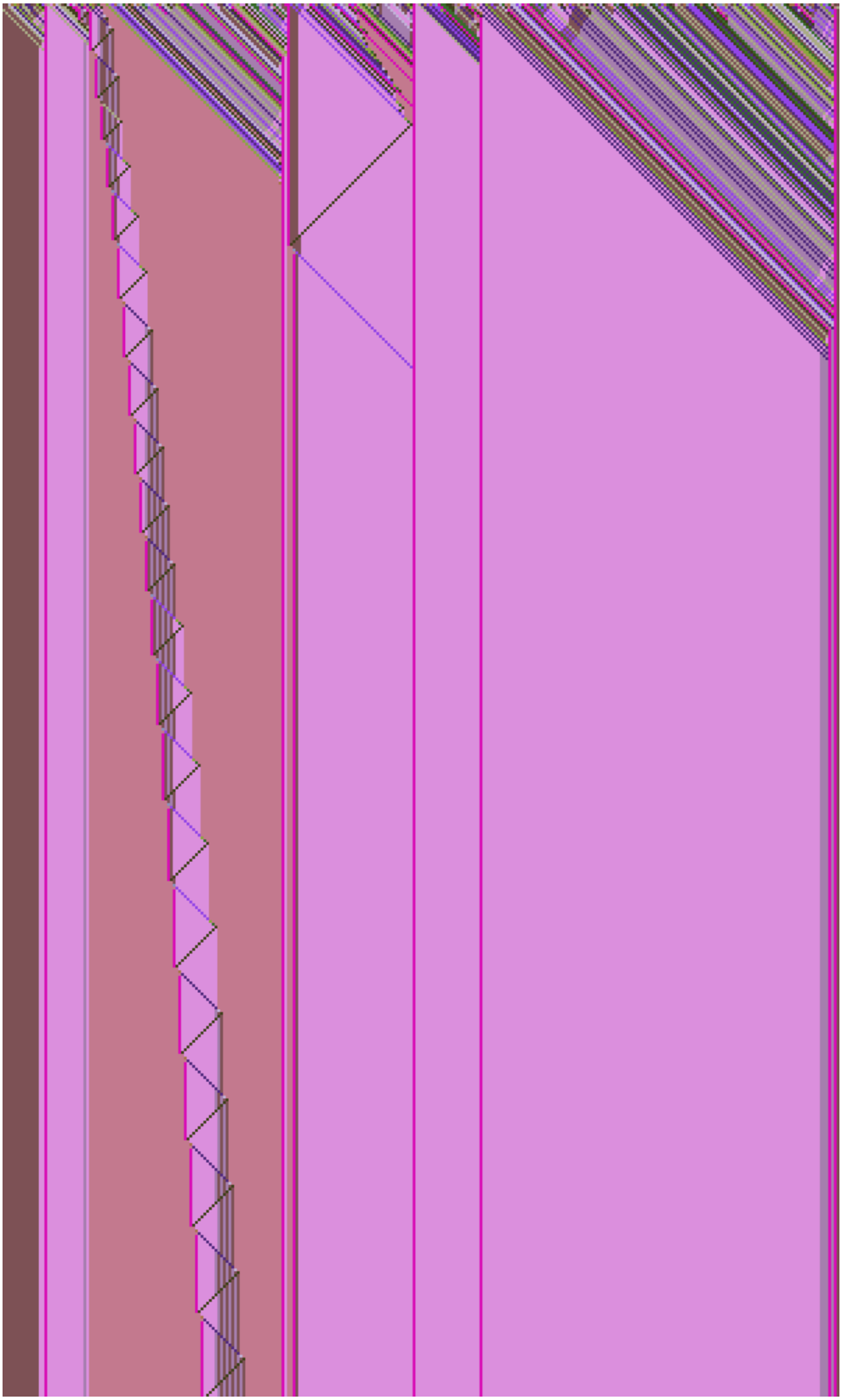


Figura 5.9: Primeras 500 evoluciones con una cadena inicial aleatoria en un arreglo de 300 celdas.

Capítulo 6

Desarrollo de un simulador

Como se vio en los capítulos 4 y 5 simular los sistemas tag y sus autómatas celulares equivalentes podría ser una tarea que lleve mucho tiempo a mano, pero la computación actual ha desarrollado plataformas que pueden ayudar a la simulación de dichos sistemas. Hasta ahora la herramienta más utilizada para simular sistemas tag y autómatas celulares es WolframAlpha desarrollada por Wolfram en su propio lenguaje llamado Mathematica. No hay una plataforma dedicada a la simulación y análisis de los sistemas tag y sus autómatas celulares equivalentes por lo que, resultado de esta investigación, se puede proponer un simulador dedicado a la tarea de simular sistemas tag y su análisis. Así que de manera breve se describirán el análisis, diseño e implementación del simulador utilizado para demostrar la funcionalidad del algoritmo propuesto en la sección 5.3.

6.1. Análisis

6.1.1. Requerimientos

Requerimientos Funcionales

En principio, un simulador debe reproducir el funcionamiento de un sistema tag. El simulador que se propone será capaz de reproducir el funcionamiento de cualquier sistema tag con su autómata celular equivalente. El simulador debe basarse en la naturaleza y modelos del sistema tag y los autómatas celulares para reproducirlos y con base en las salidas de este se verificará su funcionamiento.

El simulador será capaz de cumplir con los requerimientos funcionales y no funcionales que se especifican a continuación.

- **RF1**. Introducción de la definición de un sistema tag cualquiera, generar el autómata celular equivalente y producir las transiciones a partir de una cadena inicial.
- **RF2.** El usuario debe poder exportar un archivo JSON que contenga la definición del autómata celular que simule algún sistema tag y abrir cualquier archivo exportado.
- **RF3.** La cadena inicial puede ser especificada por el usuario o generarse una aleatoria según una longitud dada por el usuario.
- **RF4.** Las transiciones deben poder ser exportadas como un archivo de texto.
- **RF5**. Las transiciones deben ser reproducidas en un espacio bidimensional para su visualización.
- **RF6.** La visualización gráfica de las transiciones se podrá exportar como un archivo de imagen (png, jpg, tiff).

- **RF7.** El simulador proporcionará valores de análisis de sistemas dinámicos como la entropía y frecuencias de los símbolos y hacer una gráfica para cada uno.

Requerimientos no Funcionales

- **RNF1.** Para la visualización gráfica se podrá cambiar la escala de las figuras, con o sin bordes que representan cada carácter.
- **RNF2.** Debe ser posible cambiar el color asignado a cada símbolo del alfabeto del sistema tag o el del autómata celular.
- **RNF3.** La interfaz gráfica debe ser limpia y fácil de manejar.
- **RNF4.** Debe ser eficiente al momento de realizar la simulación gráfica.
- **RNF5.** El lenguaje de programación a utilizar será Java con el framework Processing.

6.1.2. Reglas de Negocio

- **RN1.** La definición del sistema tag se debe introducir utilizando una sintaxis específica.
- **RN2.** La extensión del arreglo de celdas y el número de evoluciones simuladas deben ser finitos y especificados por el usuario.

6.1.3. Arquitectura del simulador

El diagrama de bloques para el simulador se presenta en la siguiente figura:

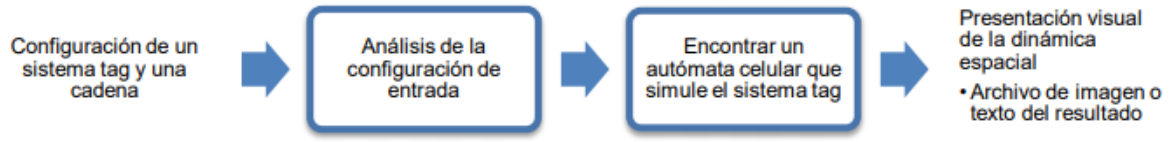


Figura 6.1: Diagrama de bloques del simulador.

6.1.4. Tecnologías utilizadas

Java

Java es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995 por Sun Microsystems. Hay muchas aplicaciones y sitios web que no funcionarán a menos que tenga Java instalado y cada día se crean más. Java es rápido, seguro y fiable. Desde portátiles hasta centros de datos, desde consolas para juegos hasta supercomputadoras, desde teléfonos móviles hasta Internet, Java está en todas partes.

Processing

Processing es un lenguaje de programación y entorno de desarrollo integrado de código abierto basado en Java, de fácil utilización, y que sirve como medio para la enseñanza y producción de proyectos multimedia e interactivos de diseño digital. Fue iniciado por Ben Fry y Casey Reas, ambos miembros de Aesthetics and Computation Group del MIT Media Lab dirigido por John Maeda. Al programar en Processing, todas las clases adicionales definidas serán tratadas como clases internas cuando el código se traduce en puro Java antes de compilar. Esto significa que el uso de variables estáticas y métodos de las clases está prohibido a menos que se indique específicamente a Processing que quiere el código en modo puro Java.

G4P (Librería)

La biblioteca G4P proporciona una amplia colección de controles GUI 2D para Processing así como el manejo de ventanas múltiples. Esta librería es extensa pero sencilla de utilizar.

Grafica (Librería)

Gráfica es una biblioteca de graficado, simple y configurable para Processing. Con esta, puede crear fácilmente gráficas 2D que disfrutarán de las capacidades interactivas completas de Processing. Puede crear gráficas simples, diagramas de dispersión e histogramas.

6.2. Diseño

Aunque los requerimientos no representan un software muy robusto es necesario definir al menos los diagramas de clases y casos de uso.

6.2.1. Diagrama de clases

El diagrama de clases describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), y las relaciones entre los objetos.

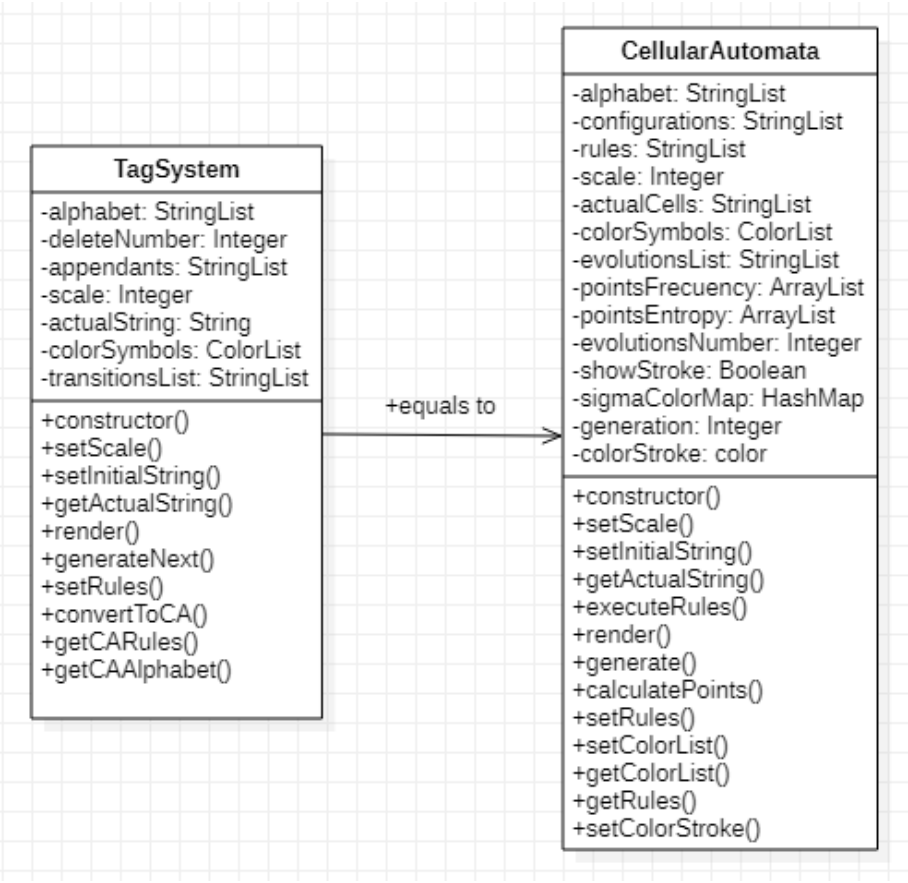


Figura 6.2: Diagrama de clases de TagSystem (Sistema Tag) y CellularAutomata (Autómata Celular).

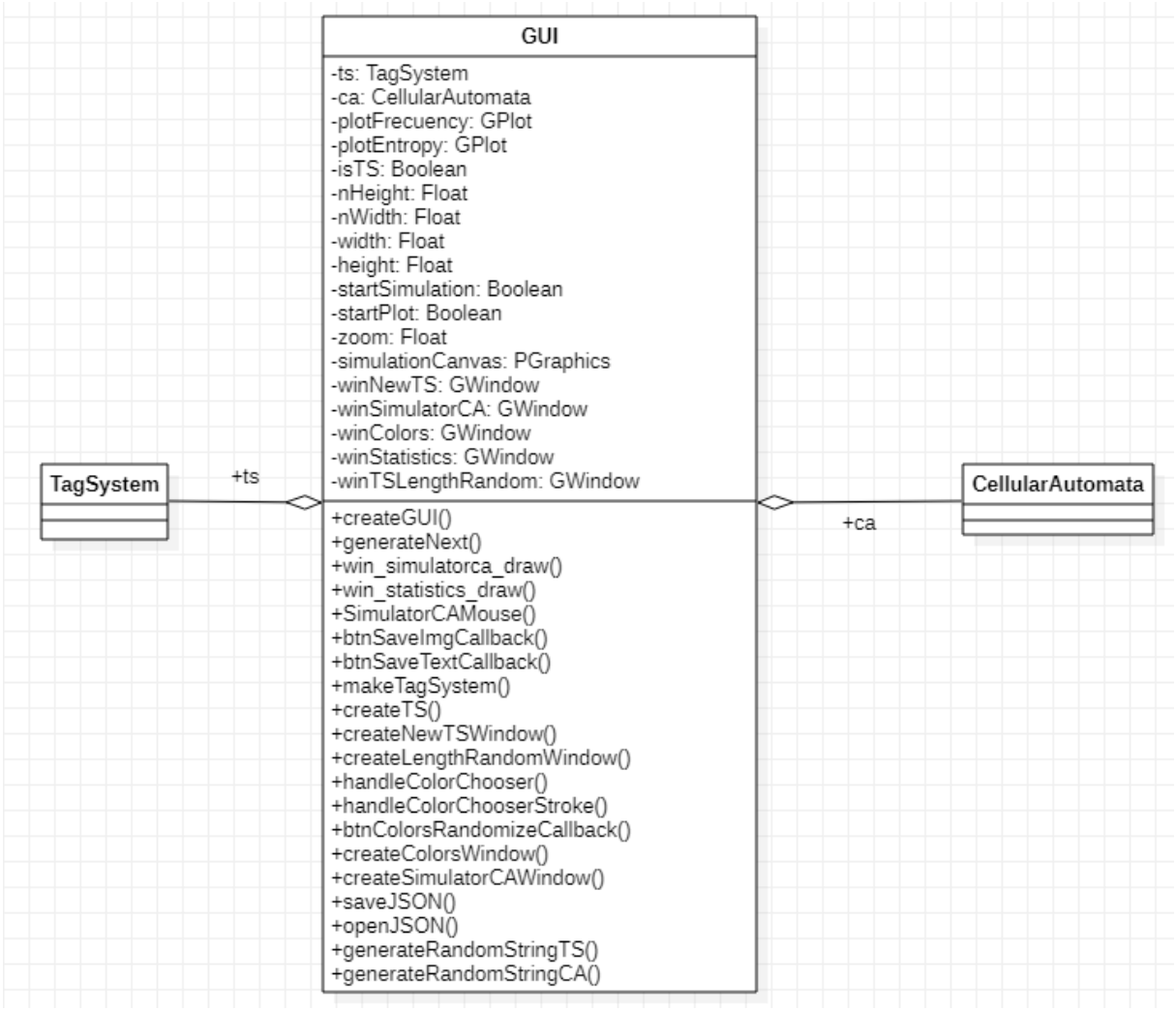


Figura 6.3: Diagrama de clases del simulador.

6.2.2. Diagrama de Casos de Uso

El diagrama de casos de uso define la naturaleza de un caso de uso; sin embargo, una notación gráfica puede solo dar una vista general simple de un caso de uso o un conjunto de casos de uso.

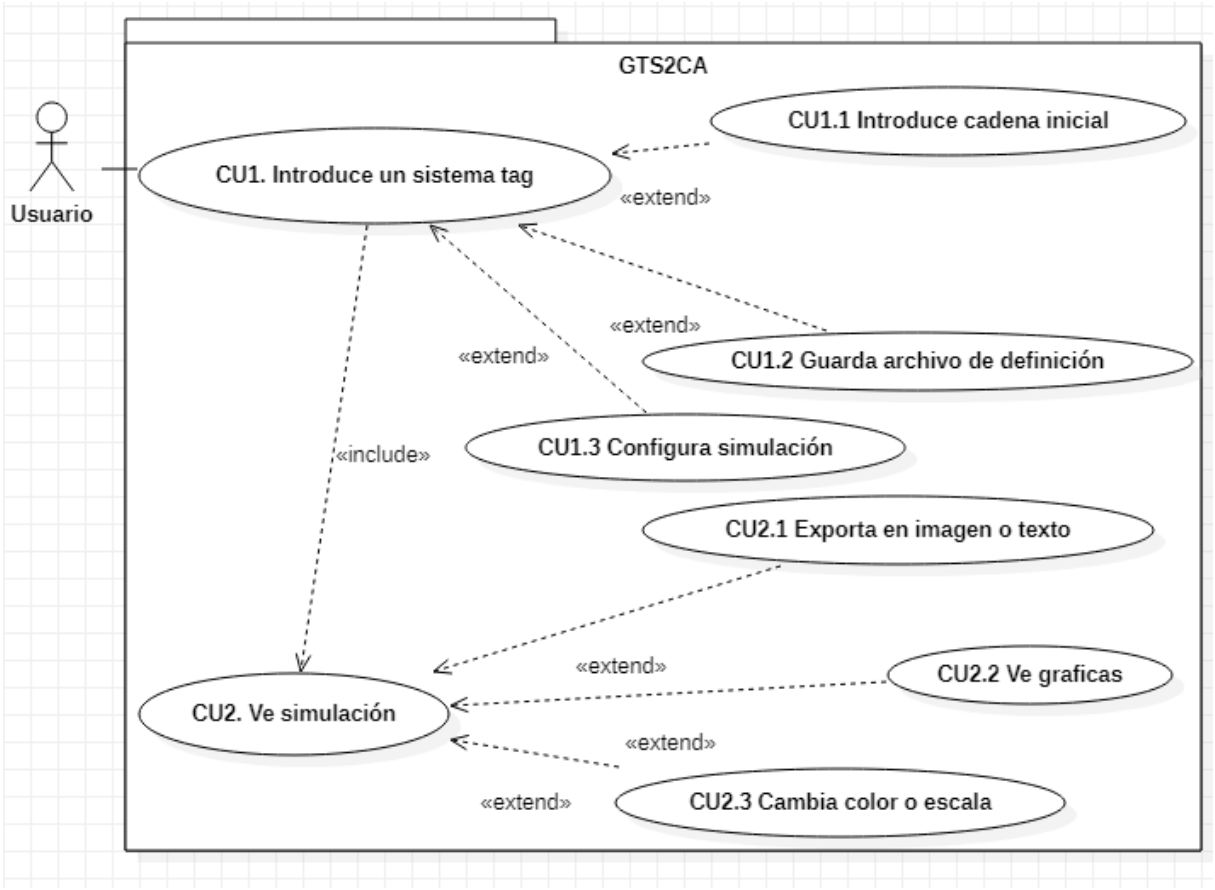


Figura 6.4: Diagrama de casos de uso del simulador.

- **CU1. Introduce un sistema tag.** El usuario final introduce la definición de un sistema tag en una ventana que recibirá los dos datos: las reglas de producción y el número de borrado. También será posible que abra un archivo JSON de un sistema tag guardado previamente.

    - **CU1.1 Introduce cadena inicial.** En una caja de texto, el usuario introducirá una cadena inicial por tres métodos: escribir una cadena inicial arbitraria, generar una cadena aleatoria con el alfabeto del sistema tag o generar una cadena inicial aleatoria con el alfabeto del autómata celular.

    - **CU1.2 Guardar archivo JSON de definición.** Después de introducir una definición de sistema tag el usuario puede guardar un archivo JSON con

la definición del sistema tag y el autómata celular equivalente.

- **CU1.3 Configura simulación**. El usuario introducirá el número de evoluciones a generar, la extensión del arreglo de celdas.

- **CU2. Ve simulación.** Una vez introducida una definición de sistema tag y una cadena inicial, se podrá iniciar la simulación, la cual tendrá lugar en una ventana de visualización gráfica.

  - **CU2.1 Exporta en imagen o texto.** El usuario puede exportar la simulación que se está visualizando en un archivo de texto o imagen, tiene la libertad de nombrarlos y elegir la ubicación del archivo en su sistema de archivos.
  - **CU2.2 Ve gráficas.** Una vez iniciada la simulación, el usuario puede visualizar las gráficas de entropía y frecuencia de cada símbolo del autómata celular en una gráfica sencilla a lo largo de las evoluciones simuladas.
  - **CU2.3 Cambia color o escala.** Antes o durante la simulación el usuario puede cambiar los colores de cada símbolo del autómata celular, cambiar la escala de cada celda, elegir si cada celda tendrá bordes o no.

6.3. Implementación

Para mostrar el resultado de la implementación de los requisitos funcionales, requisitos no funcionales, reglas de negocio y casos de uso, se mostrarán las pantallas de la interfaz gráfica del simulador GTS2CA con la explicación de cada componente de esta.

6.3.1. Pantalla inicial

La pantalla principal del simulador es la primera ventana que se mostrará al momento de abrir el programa GTS2CA, la cual contiene accesos a todas las funcionali-

dades principales de este.

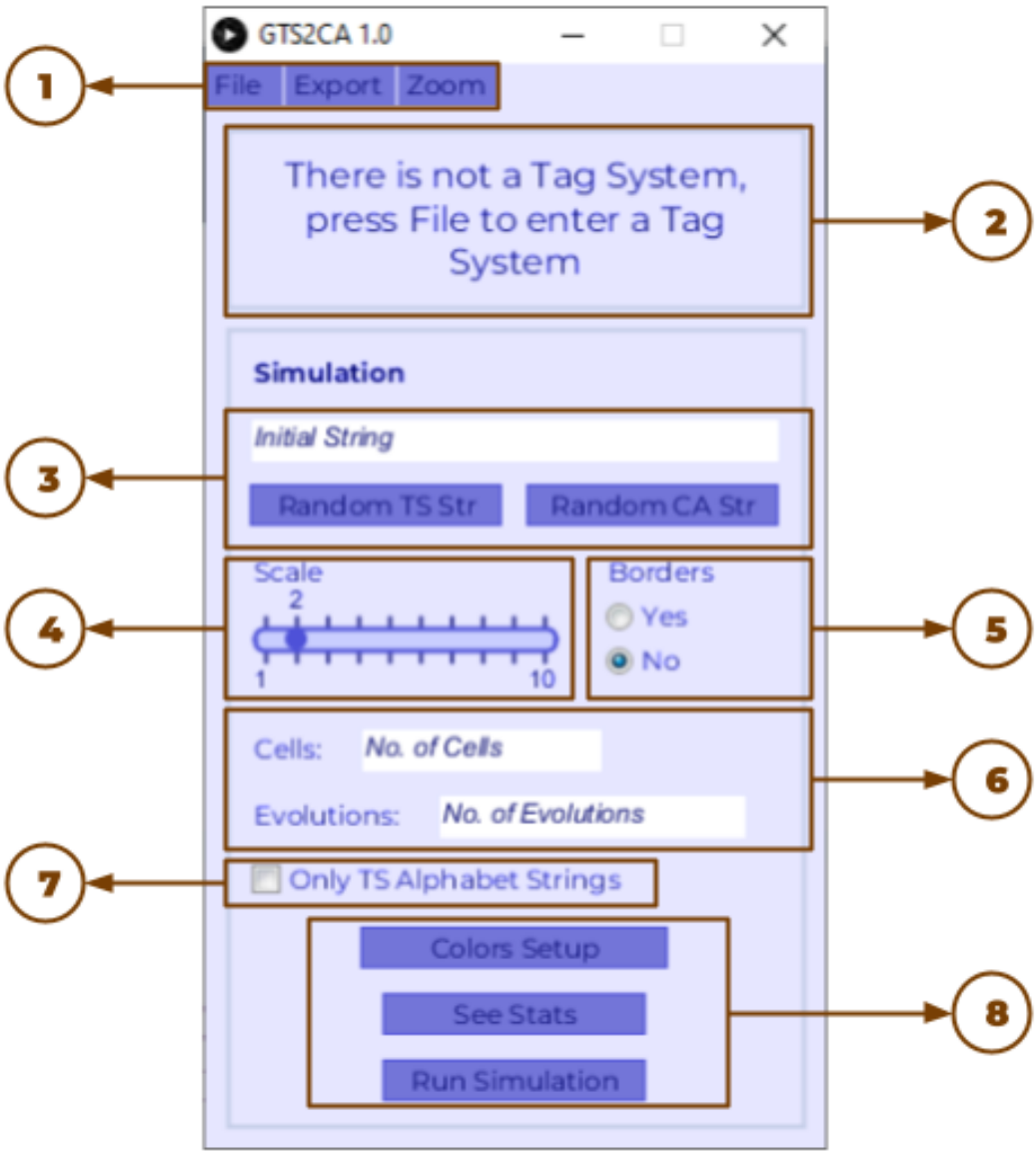


Figura 6.5: Pantalla principal del simulador GTS2CA.

1. Panel de pestañas

2. Mensaje principal

3. Caja de introducción de cadena inicial

4. Configuración de escala

5. Configuración de visualización de bordes

6. Configuración del espacio de celdas y evoluciones

7. Casilla para activar la visualización solo de cadenas del sistema tag

8. Grupo de botones de simulación

6.3.2. Panel de Pestañas

En la parte superior izquierda de la pantalla principal se encuentra este pequeño panel que consta de tres pestañas en las cuales se despliegan un conjunto de botones por cada una referente a funciones específicas que se describen a continuación.

Figura 6.6: Pestaña "File" del simulador GTS2CA.

- El botón "Open" se utiliza para abrir un archivo JSON que contiene la definición de un sistema tag y su autómata celular equivalente. Al presionarlo, se abrirá una ventana para buscar a través del directorio de archivos.

- El botón "New" abrirá una ventana donde se introducirán los datos del sistema tag (reglas de producción y número de borrado). Cada regla de producción se introduce con una sintaxis específica: σ_i -> A_i. Las reglas de producción se separan con comas: σ_1 -> A_1, σ_2 -> A_2, ..., σ_n -> A_n.

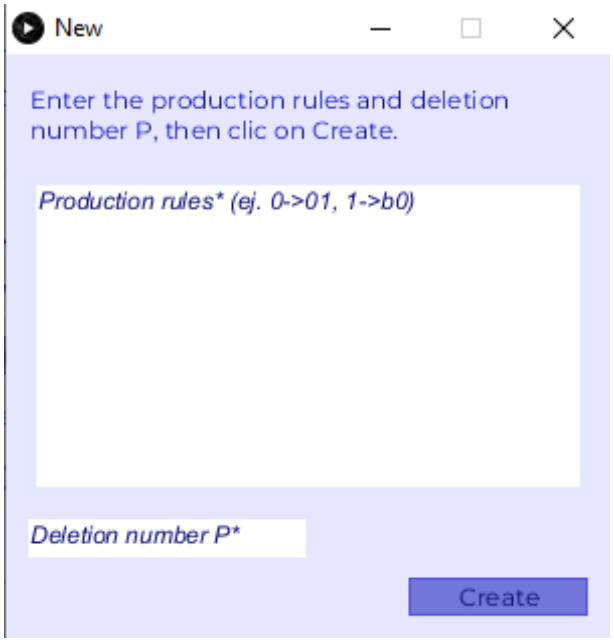


Figura 6.7: Ventana "New" del simulador GTS2CA.

- El botón "Save" se utiliza para guardar en un archivo JSON, la definición de un sistema tag y su autómata celular equivalente. Al presionarlo, se abrirá una ventana para seleccionar la carpeta y el nombre con el que se guardará. La estructura del archivo JSON guardado será la siguiente:

Figura 6.8: Estructura del archivo JSON guardado.

Las llaves ”ca” y ”ts” contienen la definición del autómata celular y el sistema tag en arreglos. En la definición de ca el contenido es:

- ”Colores”: arreglo de los colores asignados a cada símbolo del alfabeto del autómata celular.
- ”Sigma”: arreglo del alfabeto del autómata celular.
- ”Rules”: arreglo de las reglas de transición del autómata.

La definición de ”ts” es la siguiente:

- ”P”: número de borrado del sistema tag.
- ”Sigma”: arreglo del alfabeto del sistema tag.
- ”Rules”: arreglo de las reglas de producción del sistema tag.

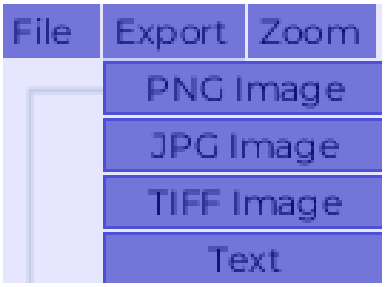


Figura 6.9: Pestaña “Export” del simulador GTS2CA.

La pestaña “Export” despliega cuatro botones de los cuales los primeros tres son para exportar una imagen en los formatos: PNG, JPG y TIFF, de la simulación. El cuarto botón es para exportar un archivo de texto CSV que contiene cada celda de cada evolución, separadas por comas.

Figura 6.10: Pestaña “Zoom” del simulador GTS2CA.

La pestaña “Zoom” contiene siete opciones de acercamiento para visualizar la imagen de la simulación.

6.3.3. Mensaje principal

A continuación se tiene una zona donde hay un mensaje principal el cual indica si hay un sistema tag cargado al programa o no lo hay.

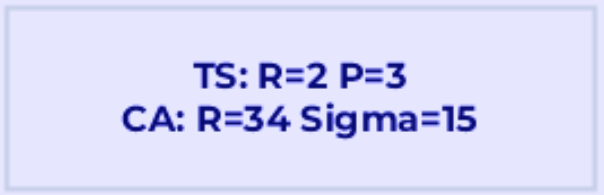


Figura 6.11: Mensaje principal cuando hay un sistema tag cargado.

Cuando hay un sistema tag cargado en el simulador el mensaje principal contiene el número de reglas de producción y número de borrado de este y debajo el número de reglas de transición y número de símbolos en el alfabeto del autómata celular equivalente.

6.3.4. Caja de introducción de cadena inicial

La caja de texto que se muestra en esta zona sirve para introducir una cadena inicial arbitraria o bien, con los botones *Random TS Str* o *Random CA Str* se pueden generar cadenas aleatorias utilizando el alfabeto del sistema tag solamente o el de su autómata celular equivalente, respectivamente.

Para generar una cadena aleatoria utilizando solamente el alfabeto del sistema tag, el programa pedirá, mediante una ventana, introducir un número que será la extensión de dicha cadena para generarla.

6.3.5. Configuración de escala

Para configurar el tamaño de cada celda de la simulación gráfica, se tiene un deslizador (Slider) que aumenta o disminuye dicha escala. La modificación de la escala también modifica el tamaño de la imagen de exportación debido a que se cambia el tamaño del espacio de simulación proporcionalmente a la escala. El deslizador tiene un rango de 1 a 10 donde cada valor representa el tamaño de la celda en pixeles. Por defecto se establece en 2 pixeles.

6.3.6. Configuración de visualización de bordes

Del lado derecho del deslizador para configurar la escala, se encuentran unos botones radiales (RadioButton) agrupados como un botón palanca (ToggleButton). Al momento de configurar la visualización de bordes para que cada celda sea rodeada por un borde lineal solo hay dos estados posibles: con bordes o sin bordes. Así que este botón es justo uno de los que ofrece la solución de elegir solo un estado exclusivo del otro. Al elegir "Yes" se activa la visualización de bordes y No para desactivarla. Por defecto el valor es "No".

6.3.7. Configuración del espacio de celdas y evoluciones

Esta zona provee de dos cajas de texto (TextField) en las que la primera (Cells) se introduce el número de celdas que tendrá el autómata celular y la segunda (Evolutions) es el número de evoluciones que se simularán. Por defecto los valores en estas cajas de texto son: 300 y 1000, respectivamente.

6.3.8. Casilla para activar la visualización solo de cadenas del sistema tag

Cuando esta casilla está activada se descartarán las evoluciones que contengan símbolos auxiliares y los símbolos de control $\{L, U\}$, es decir, únicamente se visualizarán las cadenas que contengan símbolos del alfabeto del sistema tag.

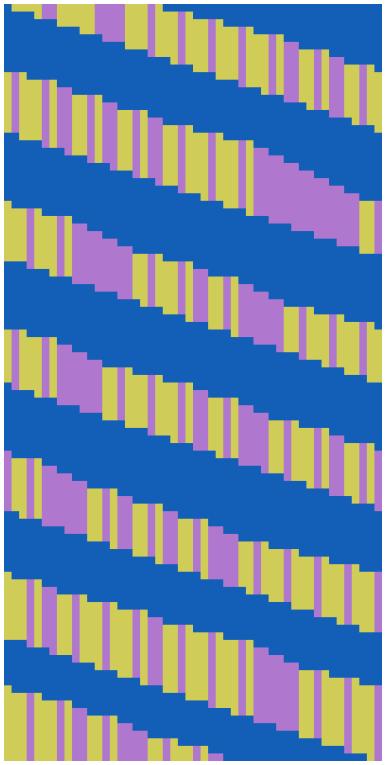


Figura 6.12: Simulación cuando está activada la visualización solo de cadenas del sistema tag.

6.3.9. Grupo de botones de simulación

En la parte inferior de la pantalla principal se encuentran tres botones relacionados con la simulación y el análisis de los datos generados.

- El botón “Colors Setup”, al presionar este botón se abrirá una ventana donde encontraremos algunas opciones de configuración de los colores de la simulación. El primero es un selector desplegable donde se selecciona el símbolo del autómata celular al cual queremos cambiar el color, justo al lado derecho de este selector se encuentra el botón “Change” que abrirá una ventana para seleccionar el color. Debajo se encuentra la sección de bordes con otro botón “Change” el cual permite cambiar el color de los bordes. El penúltimo botón es “Randomize colors” el cual cambia todos los colores de todos los símbolos

de manera aleatoria. Al terminar de configurar los colores se presiona el botón "Ok".

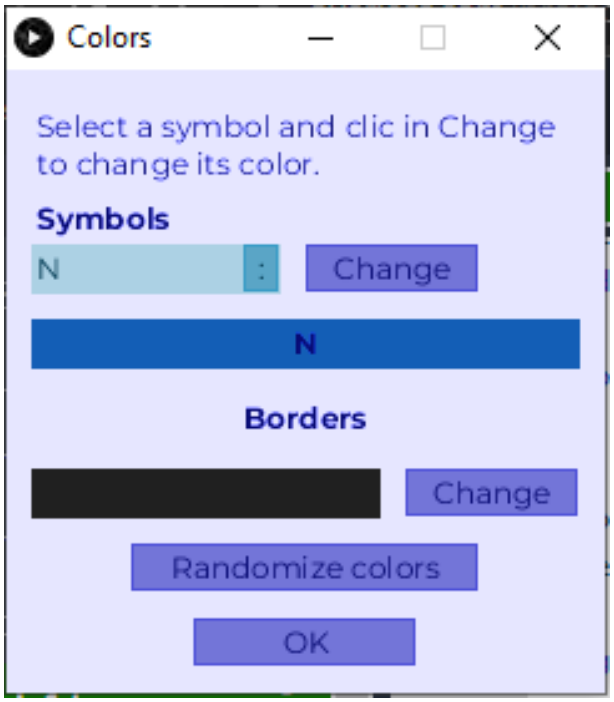


Figura 6.13: Ventana "Colors" del simulador GTS2CA.

- El botón "See Stats" se utiliza para abrir la ventana de estadísticas de la simulación. La ventana de estadísticas muestra una gráfica de frecuencia en las que se visualiza la frecuencia con la que aparece cada símbolo y debajo, una segunda gráfica en la que se muestra la entropía de Shannon de cada evolución. La entropía de Shannon se calcula utilizando la formula:

$$H = \sum_{i=1}^{k} -\frac{n_i}{N} \log_2 \frac{n_i}{N}$$

Donde:

- H es la entropía de Shannon.
- k es el número de símbolos en el alfabeto.

- n_i es el número de veces que aparece el i-esimo símbolo en la cadena.
- N es el número de símbolos total en la cadena.

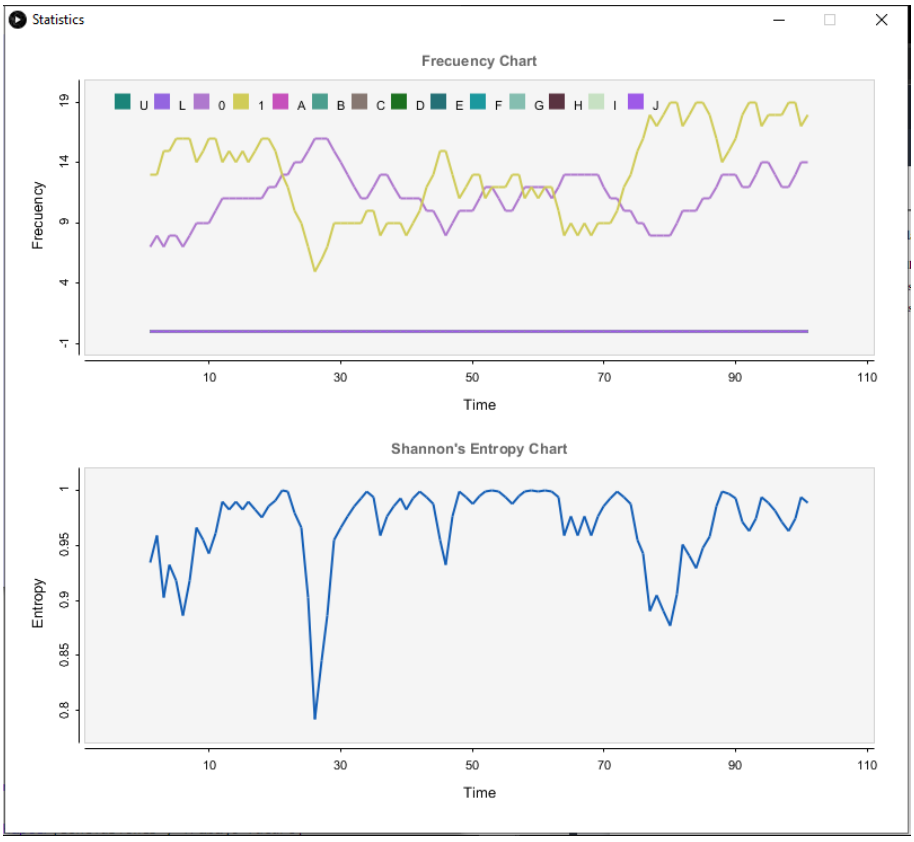


Figura 6.14: Ventana de estadísticas del simulador GTS2CA.

- El botón “Run Simulation” iniciará la simulación en una ventana de simulación. En la ventana de simulación se pueden utilizar dos eventos del ratón: arrastrar (Mouse Drag) y la rueda del ratón (Mouse Wheel). El evento arrastrar permite desplazar la imagen generada por el simulador en todas las direcciones para visualizar zonas con mayor detalle y el evento de la rueda del ratón permite aplicar un Zoom instantáneo a la imagen generada, hacia arriba es para agrandar la imagen y hacia abajo es para encogerla.

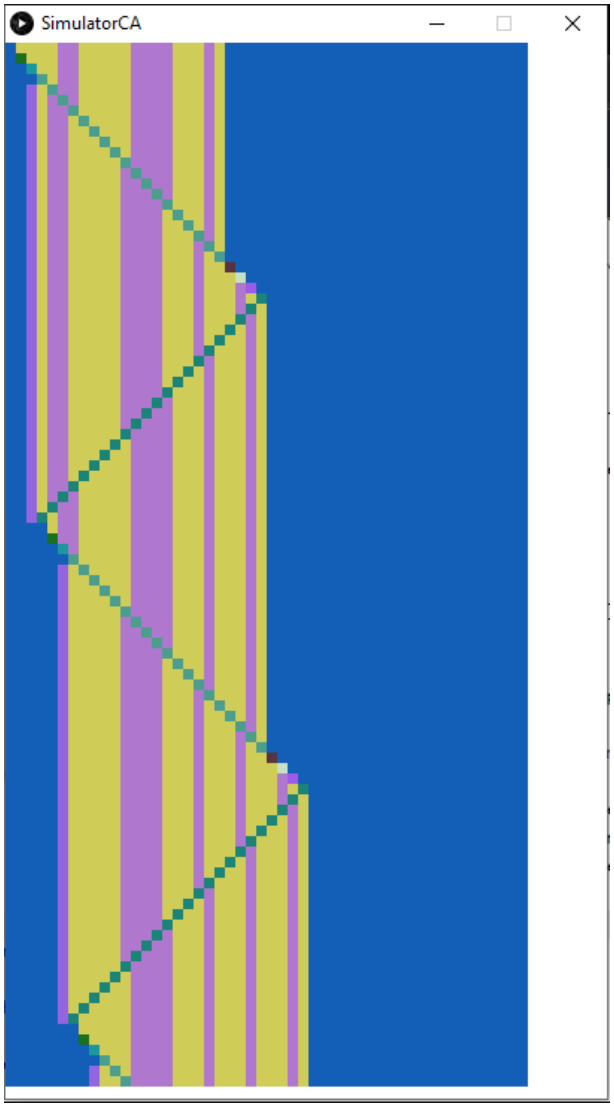


Figura 6.15: Ventana de simulación.

El código fuente del simulador se encuentra en los Anexos al final este documento. En los Anexos también se tiene un enlace a un repositorio de GitHub en el cual están los archivos del código fuente que en el futuro tendrán mantenimiento y actualizaciones.

Capítulo 7

Exploración de los Autómatas Celulares que simulan Sistemas Tag

Con el desarrollo de un simulador que cumple con las necesidades de este trabajo terminal y los fundamentos teóricos desarrollados en los primeros capítulos se explorarán los autómatas celulares equivalentes a algunos sistemas tag para demostrar el funcionamiento del algoritmo propuesto en la sección 5.3 y algunas características que pueden ser observadas de estos como sistemas dinámicos discretos.

7.1. Exploración

En esta sección se presentan algunos ejemplos simulados de sistemas tag y sus autómatas celulares equivalentes. Los ejemplos a continuación son sistemas tag de alfabetos binarios o ternarios con $P = 1$, $P = 2$ o $P = 3$.

7.1.1. Sistema Tag de las Cadenas de Fibonacci

El sistema tag descrito en la sección 4.4 que produce cadenas de Fibonacci mediante:

$$0 \to 01, 1 \to 0$$

$$P = 1$$

Cuyas cadenas generadas se obtienen de la fórmula original:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 01 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Una vez introducido al algoritmo de la sección 5.3 se obtuvo un autómata celular de 23 reglas de transición y un alfabeto de 8 símbolos, a continuación muestran algunas de las simulaciones con cadenas del sistema tag y condiciones iniciales aleatorias. En la figura 7.1 se observa el mecanismo por el cual el autómata celular genera las cadenas del sistema tag. La longitud de las cadenas va en aumento porque por cada símbolo borrado se añade uno o dos. Esto se demuestra en la figura 7.2 donde se ve el crecimiento en la longitud de las cadenas del sistema tag sin visualizar los pasos intermedios.

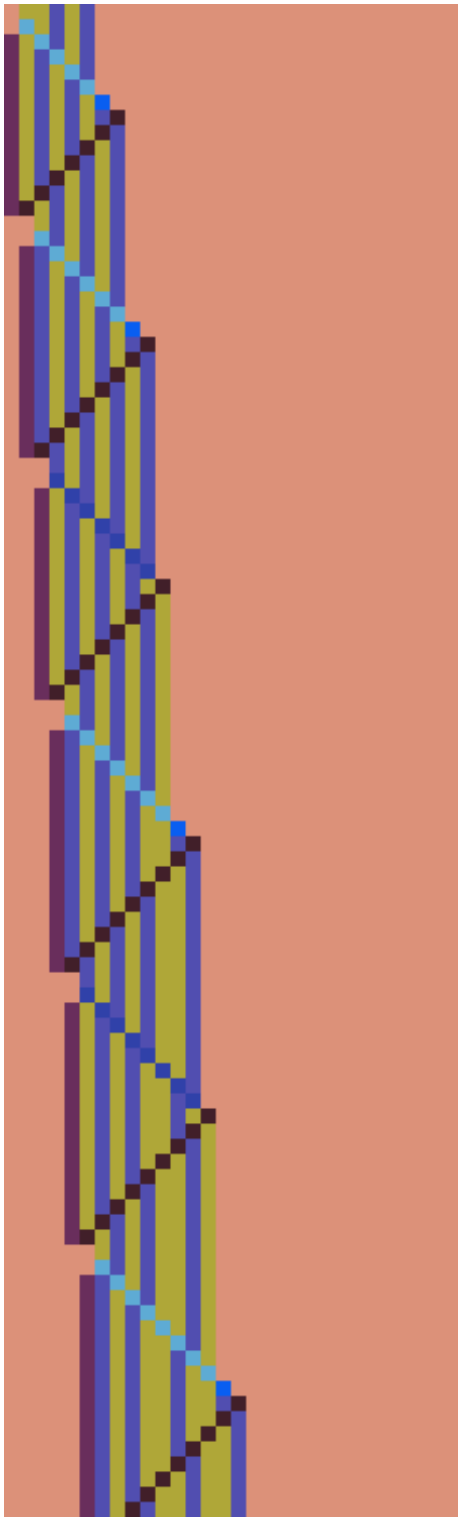


Figura 7.1: Autómata celular equivalente con cadena inicial 00101 en un arreglo de 30 celdas y 100 evoluciones.

Al quitar los pasos intermedios del autómata celular para el cálculo de las cadenas generadas por este sistema tag se obtiene:

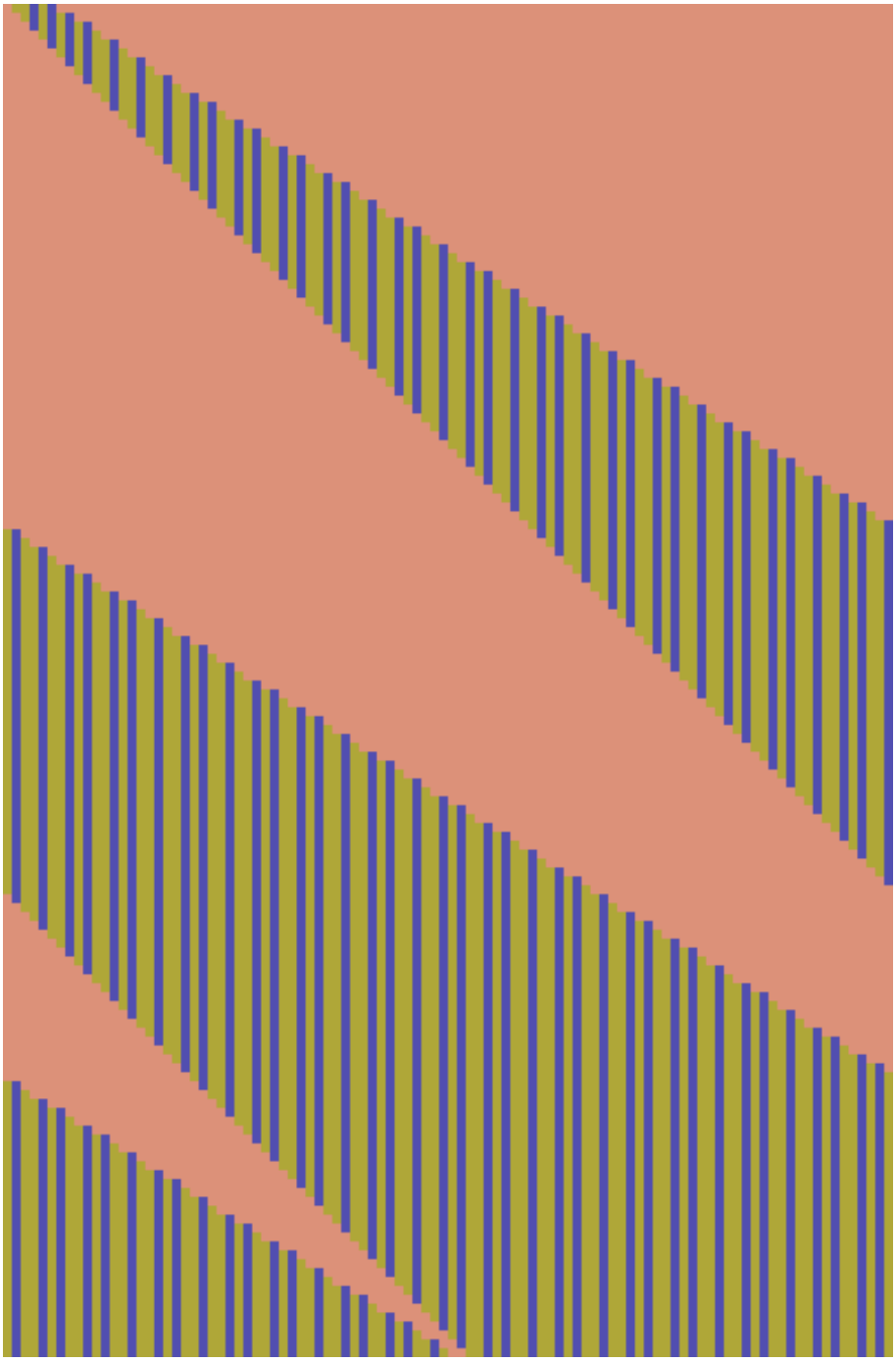


Figura 7.2: Simulación sin pasos intermedios de la cadena inicial 00101 en un arreglo de 100 celdas, 16017 evoluciones y 152 cadenas generadas.

Con condiciones iniciales aleatorias en el autómata celular se puede observar que en algunas ocasiones se notan múltiples patrones similares a los de introducir una cadena del sistema tag:

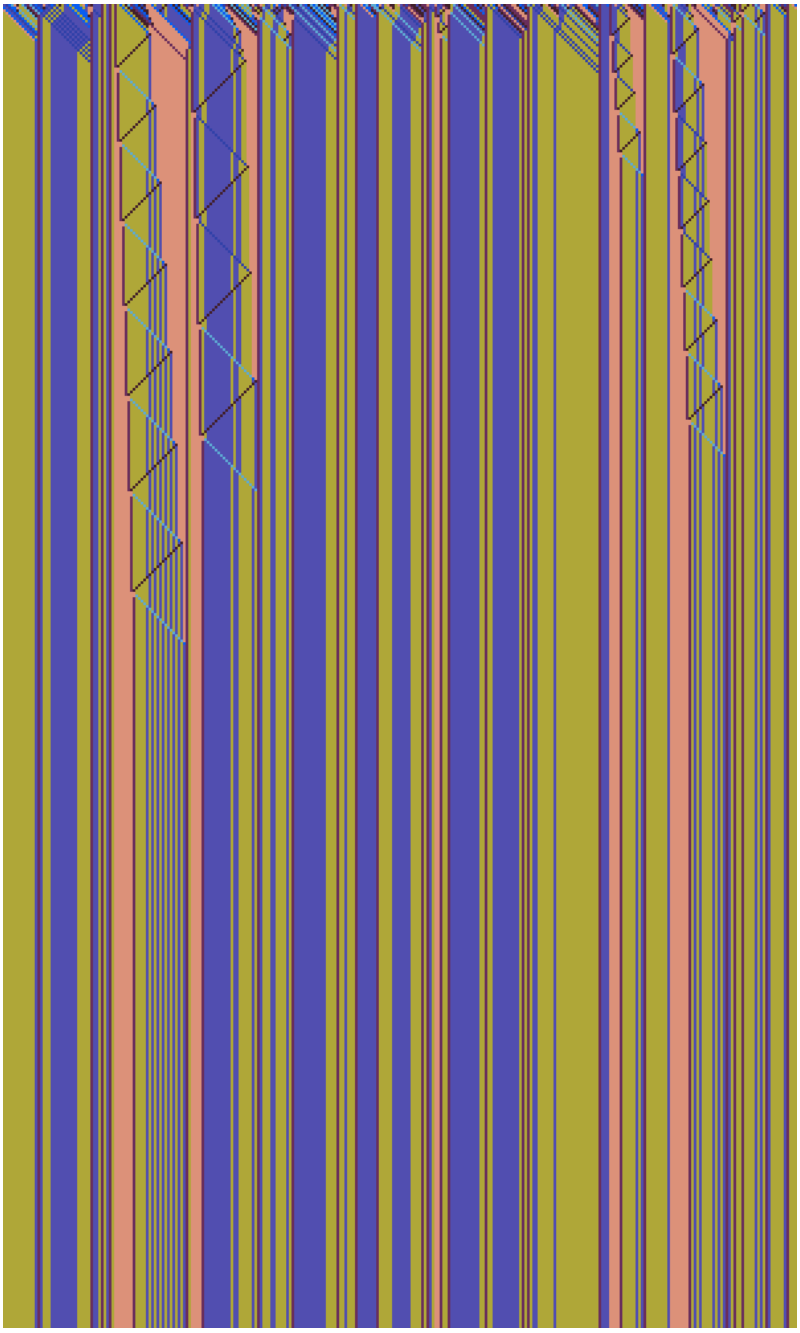


Figura 7.3: Condiciones iniciales aleatorias en el autómata celular equivalente en un arreglo de 300 celdas y 500 evoluciones.

7.1.2. Sistemas Tag con $P = 2$ y Tres Símbolos

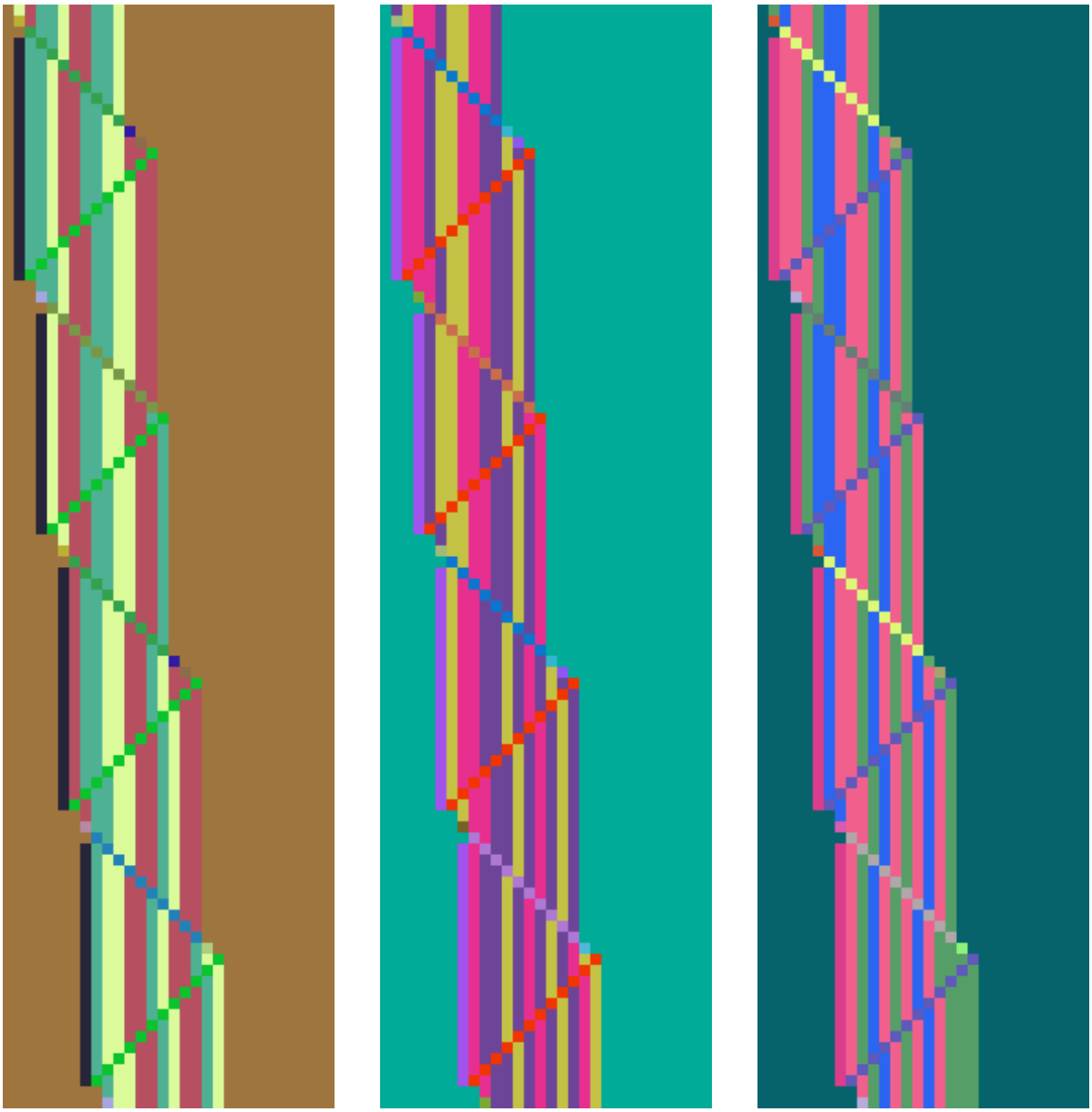
Stephen Wolfram en su artículo *After 100 Years, Can We Finally Crack Post's Problem of Tag? A Story of Computational Irreducibility, and More* (Después de 100 años, ¿podemos finalmente resolver el problema de las etiquetas de Post? Una historia de irreductibilidad computacional y más) [12] presenta algunos ejemplos de sistemas tag con $P = 2$ y $\Sigma_{TS} = \{0, 1, 2\}$. Las reglas de producción de cada uno son las siguientes:

a) $0 \to 0, 1 \to 02, 2 \to 211$

b) $0 \to 0, 1 \to 01, 2 \to 212$

c) $0 \to 0, 1 \to 22, 2 \to 102$

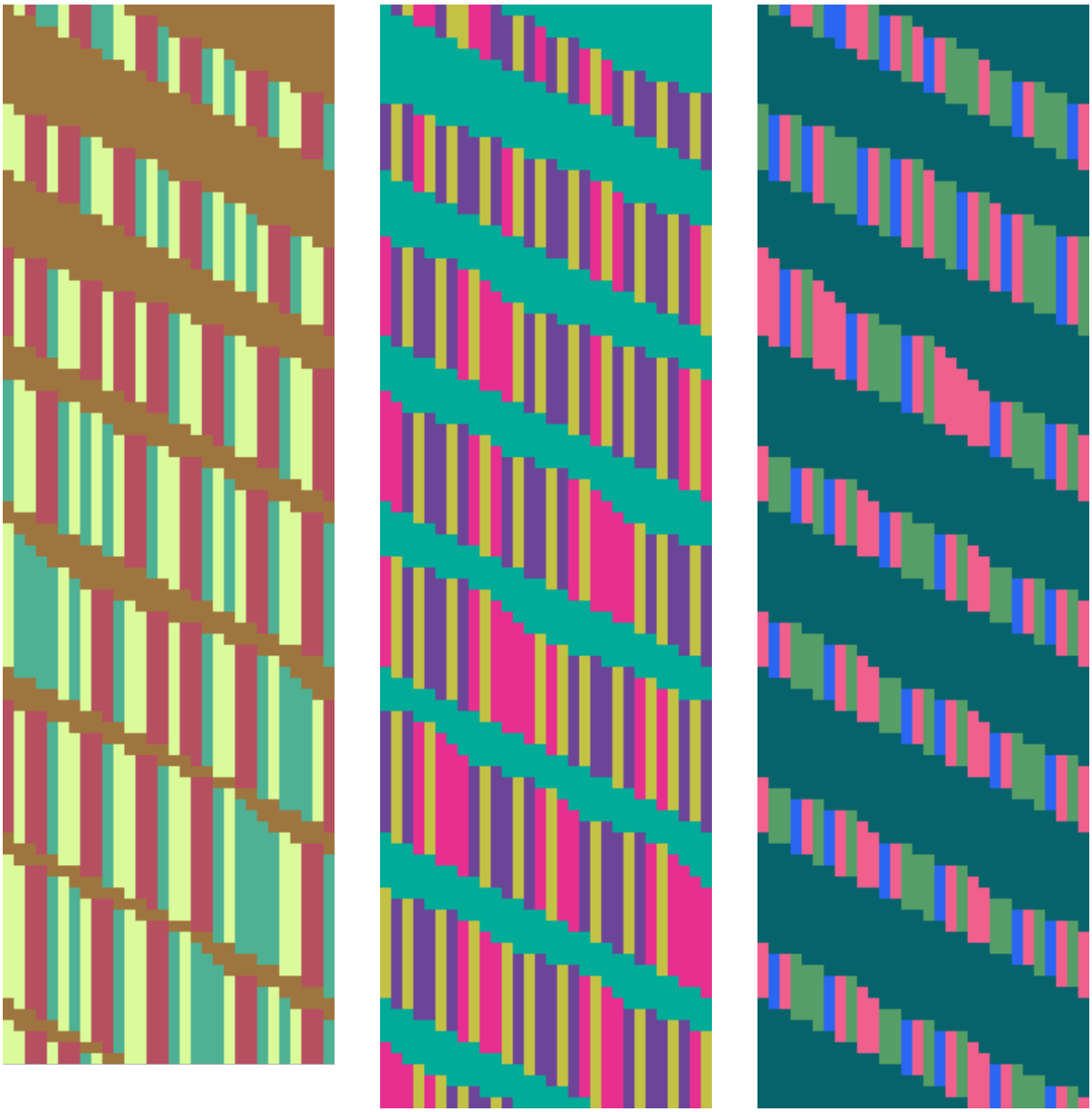
Cuyos autómatas celulares equivalentes calculados tienen características iguales: 38 reglas y 15 símbolos. Los alfabetos calculados son equivalentes, pero difieren en su comportamiento debido a que varias reglas de transición son distintas. A continuación se muestran algunas simulaciones de cada uno:



(a) (b) (c)

Figura 7.4: Autómatas celulares equivalentes con cadena inicial 2100211002 en un arreglo de 30 celdas y 100 evoluciones.

En la Figura 7.4 se puede observar que cada autómata celular hace un cálculo diferente para la misma cadena 2100211002 aunque b) y c) presentan patrones muy similares.

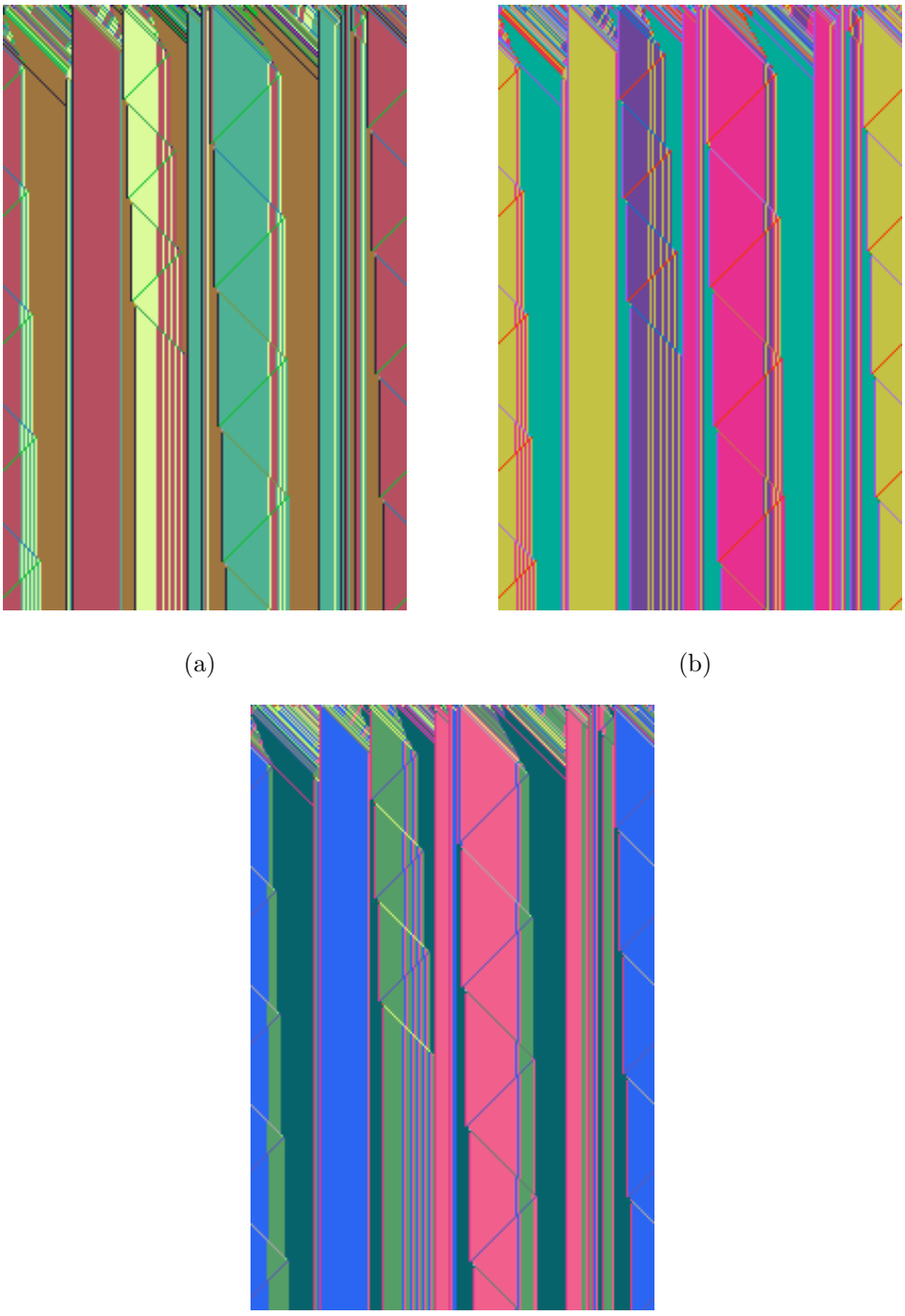


(a) (b) (c)

Figura 7.5: Cadenas generadas de cada sistema tag con la cadena inicial 2100211002 en un arreglo de 30 celdas: a) 4452 evoluciones y 95 cadenas, b) 4281 evoluciones y 100 cadenas, c) 2661 evoluciones y 100 cadenas.

En la Figura 7.5 las simulaciones muestran el cálculo que cada sistema tag realiza sobre la misma cadena 2100211002, los cuales corresponden a la generación de cadenas diferentes demostrando que los autómatas celulares que corresponden a cada

uno, en realidad generan cadenas distintas a pesar de que en los pasos intermedios muestren un comportamiento muy semejante entre sí.



(a) (b)

(c)

Figura 7.6: Cada autómata celular comparte las mismas condiciones iniciales aleatorias en un arreglo de 200 celdas y 300 evoluciones.

En las mismas condiciones iniciales aleatorias se desarrollan comportamientos muy semejantes, pero en varias estructuras los patrones son distintos, este fenómeno es posible debido a la definición de cada sistema tag mostrado porque presentan muchas similitudes.

7.1.3. Sistema Tag de la Máquina de Turing Universal de Wolfram más pequeña

En la sección 4.7 se define un sistema tag que produce los movimientos del cabezal de una máquina de Turing universal de dos estados y tres colores. Este sistema tag es de número de borrado $P = 2$ y las reglas de producción son:

$$0 \to 32, 1 \to 65, 2 \to 11, 3 \to 0, 4 \to, 5 \to 00, 6 \to 11, 7 \to 00, 8 \to ba, 9 \to ed, a \to 9999,$$
$$b \to 8, c \to 8, d \to 8, e \to 9999, f \to 8888, g \to ji, h \to ml, i \to 99hh, j \to ggoooo, k \to ggoo,$$
$$l \to ggoooo, m \to 9999h, n \to, o \to rq, p \to ut, q \to p, r \to oooo, s \to oooo, t \to oooo,$$
$$u \to p, v \to o$$

Se calculó un autómata celular de 364 reglas de transición y un alfabeto de 149 símbolos, a continuación muestran algunas de las simulaciones con cadenas del sistema tag y condiciones iniciales aleatorias:

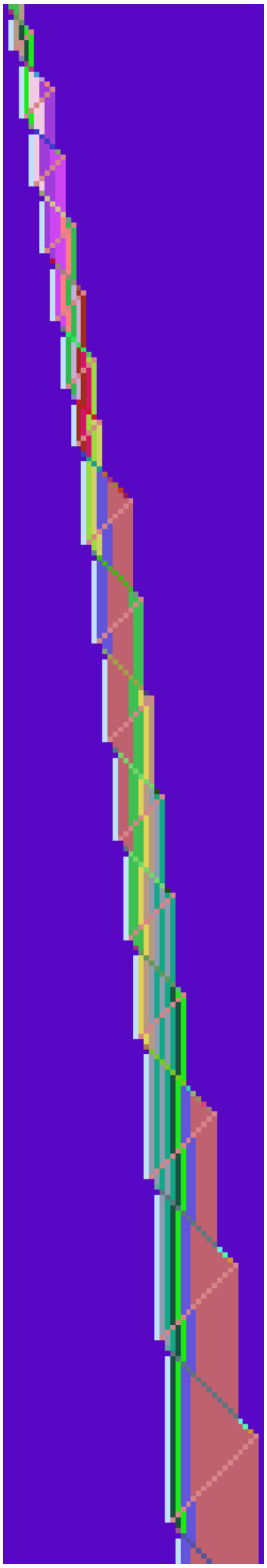


Figura 7.7: Autómata celular equivalente con cadena inicial $00i$ en un arreglo de 50 celdas y 300 evoluciones.

Al quitar los pasos intermedios del autómata celular para el cálculo de las cadenas generadas por este sistema tag se observan bloques de cadenas donde se repiten uno o dos símbolos que representan el movimiento del cabezal de la máquina de Turing en una dirección tal como se presenta en la figura 7.8:

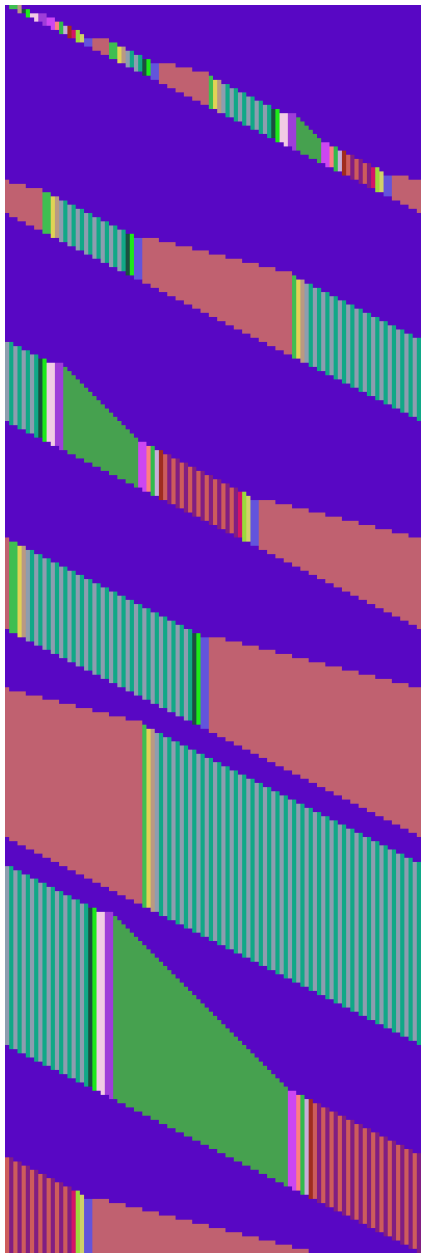


Figura 7.8: Cadenas generadas del sistema tag con la cadena inicial $00i$ en un arreglo de 100 celdas y 300 cadenas generadas en 29897 evoluciones.

Debido a que este autómata celular tiene un alfabeto amplio y las reglas de transición son mínimas comparadas con el número de reglas posibles con este alfabeto:

$$C_T = |\Sigma|^{2r+1} = 149^{2(1)+1} = 149^3 = 3,307,949$$

$$R_T = |\Sigma|^{C_T} = 149^{3,307,949}$$

Es difícil encontrar un comportamiento complejo con condiciones iniciales aleatorias, las exploraciones realizadas resultan en configuraciones homogéneas. La búsqueda de condiciones iniciales que resulten en un comportamiento complejo esta más allá de los alcances de este trabajo. Pero su funcionamiento con cadenas del alfabeto del sistema tag ha sido demostrado por la exploración realizada.

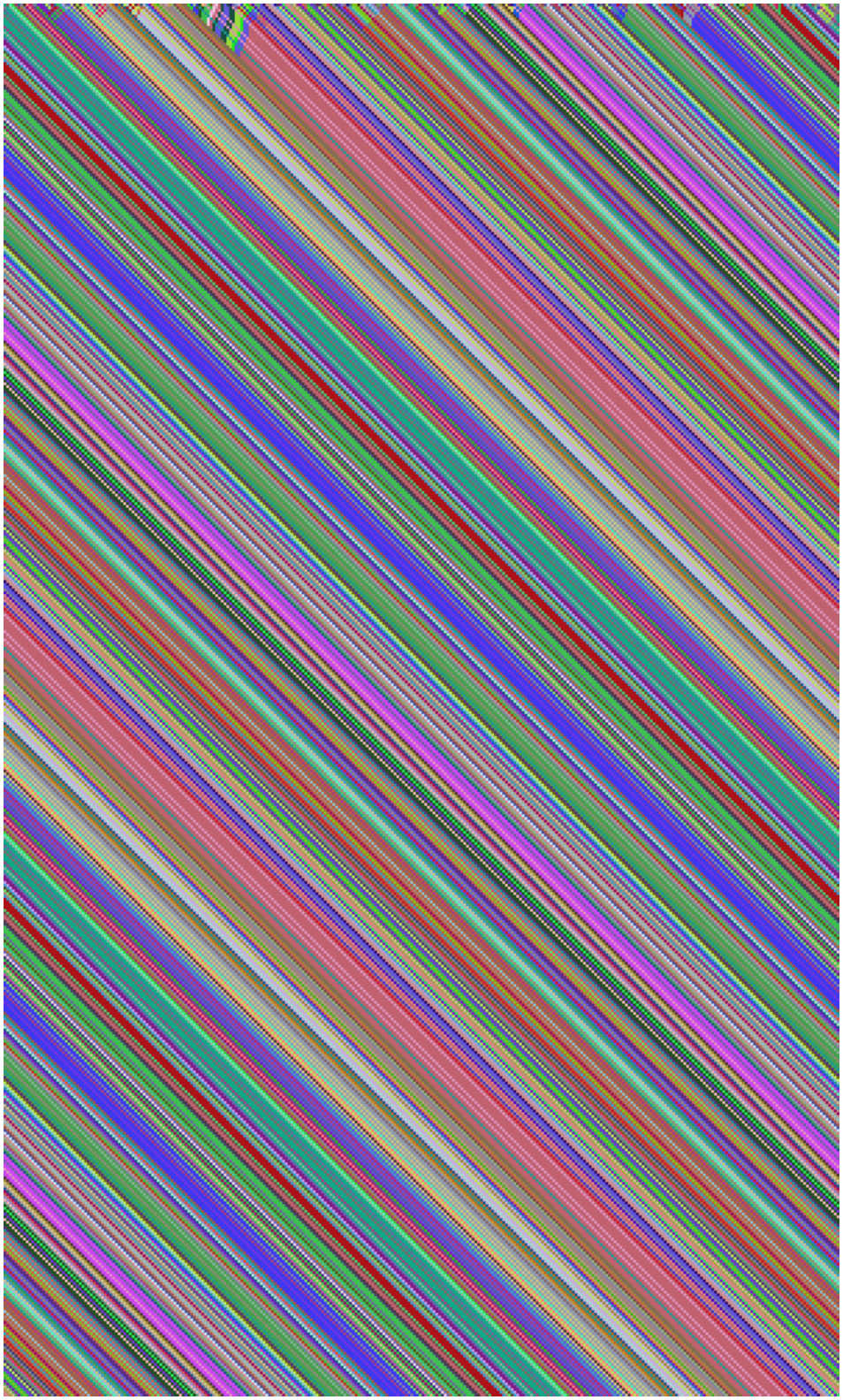


Figura 7.9: Condiciones iniciales aleatorias en el autómata celular equivalente en un arreglo de 300 celdas y 500 evoluciones.

7.2. Propiedades

7.2.1. Computación Paralela

Una de las principales ventajas de calcular autómatas celulares equivalentes a sistemas tag es que los autómatas celulares pueden ampliar el arreglo de celdas y el espacio de evolución que teóricamente pueden ser infinitos, aunque son limitados por la memoria de dispositivo haciéndolos finitos en la práctica. Esto implica que se pueden introducir múltiples cadenas en el alfabeto del sistema tag separadas por el símbolo del espacio nulo N. La configuración de las condiciones iniciales se realiza mediante la expresión regular:

$$(\Sigma_{TS}^{+}(\Sigma_{TS}^{+})^{*}N(N^{*}))^{n}$$

Donde:

- Σ_{TS} es el alfabeto del sistema tag.
- N es el símbolo del espacio nulo.
- n es el número de cadenas a calcular.

A continuación se presentan algunos ejemplos donde se aplica este método de paralelización del sistema tag:

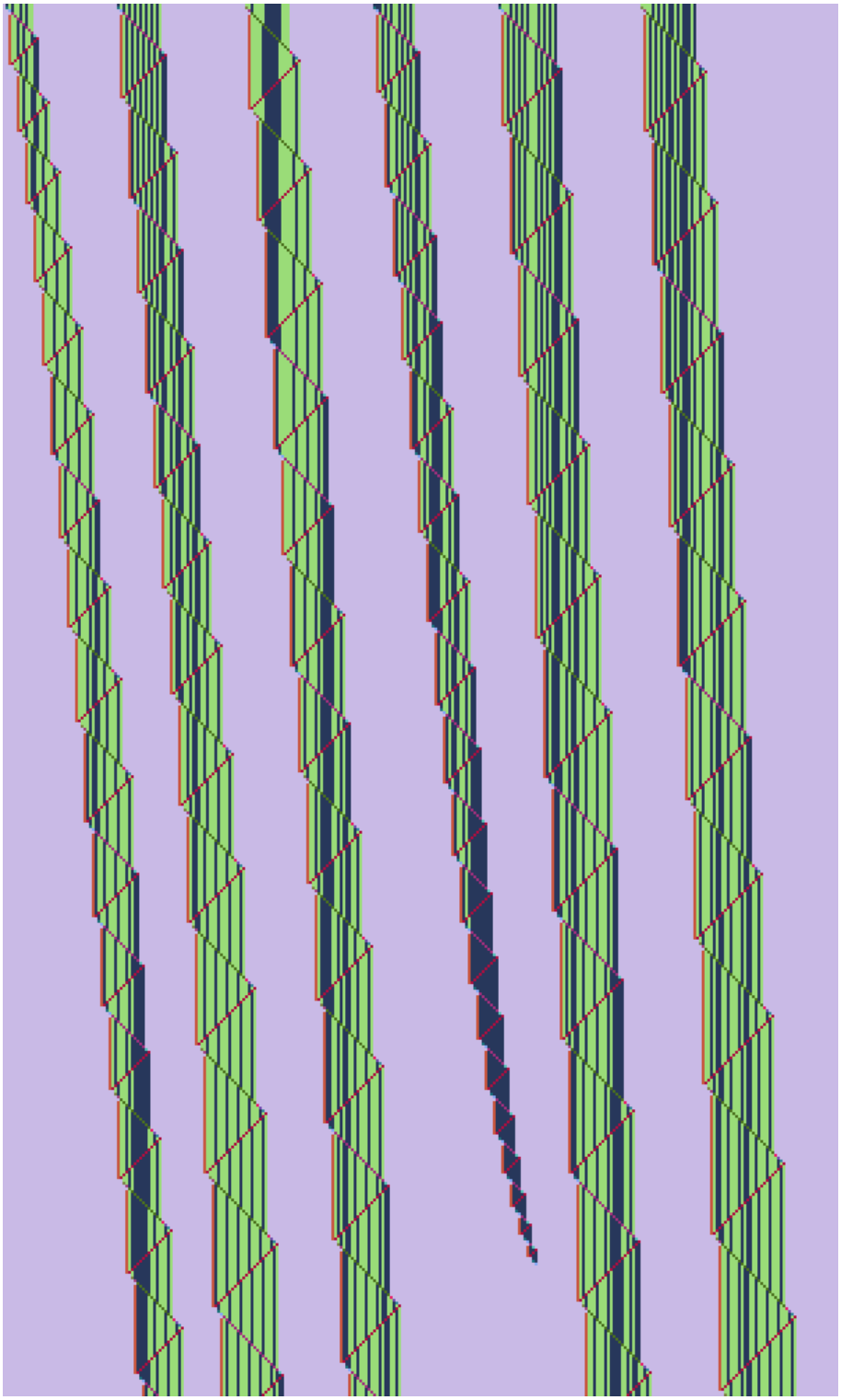


Figura 7.10: Autómata celular equivalente al sistema tag de la sección 4.8 calculando paralelamente 6 cadenas (0101101011, 0101010101010101, 1101111000000111, 000101011010101, 011101010101110101010, 111101010100101000 10).

Figura 7.11: Autómata celular de la subsección 7.1.3 calculando paralelamente 5 cadenas ($c8ref6o7lg$, $5o557fq76s$, $lkcdo2r9eq$, $0npe73200i$, $gfbjtienh9$).

7.2.2. Gráficas de Frecuencia y Entropía de Shannon

Las gráficas de frecuencia son aquellas que muestran el número de veces que aparece un símbolo en cada evolución. Es decir, es una gráfica bidimensional donde en el eje x es el número de evoluciones y el eje y es la frecuencia de los símbolos.

La entropía de Shannon calcula la cantidad de información que hay en una cadena de símbolos, el número resultante indica el número de bits que se necesitan para codificar dicha información. Se calcula con la fórmula:

$$H = \sum_{i=1}^{k} -\frac{n_i}{N} \log_2 \frac{n_i}{N}$$

Donde:

- H es la entropía de Shannon.
- k es el número de símbolos en el alfabeto.
- n_i es el número de veces que aparece el i-esimo símbolo en la cadena.
- N es el número de símbolos total en la cadena.

En la gráfica de la entropía de Shannon se observa como cambia H por cada evolución. A continuación se muestran algunos ejemplos de estas gráficas.

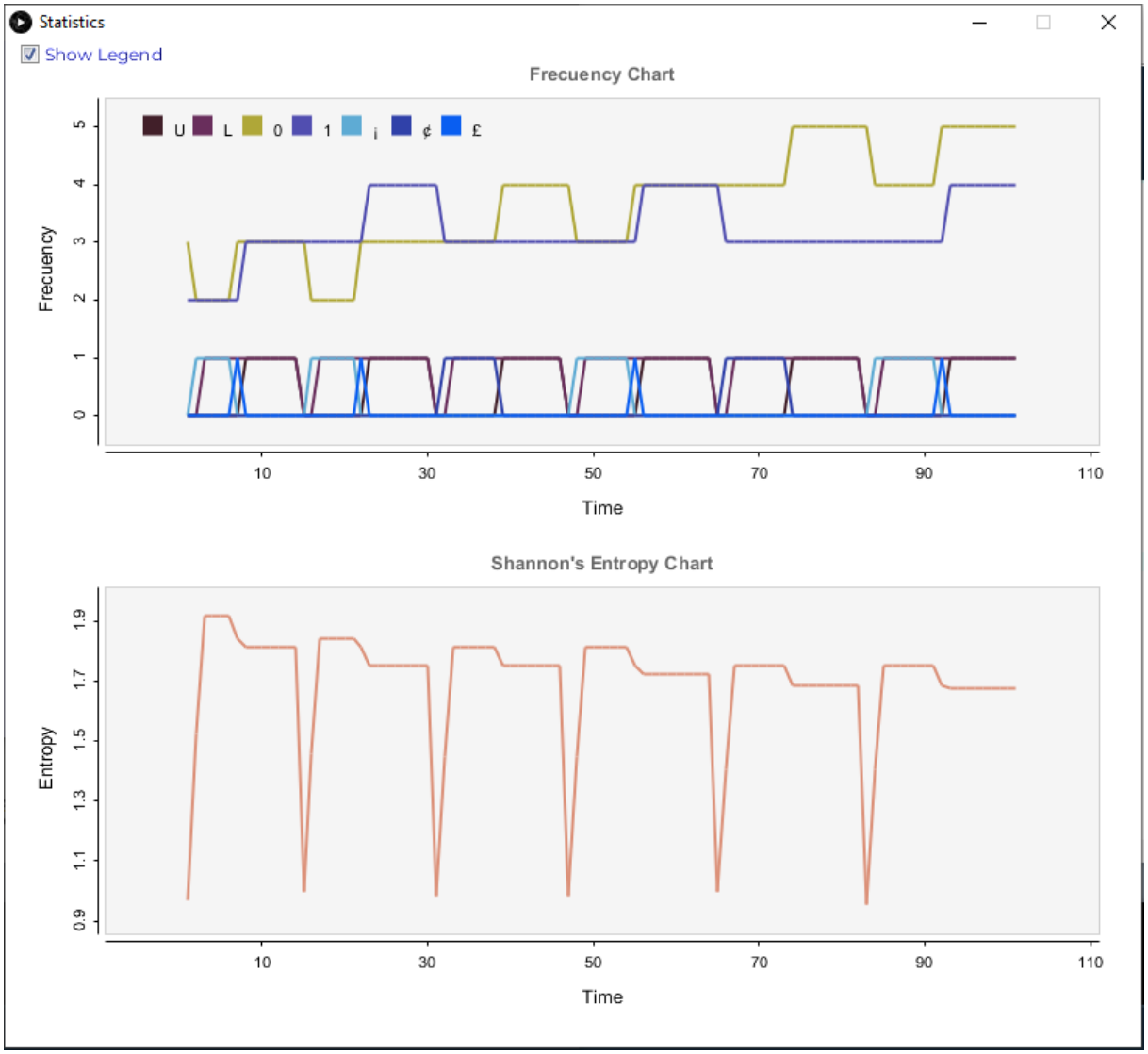


Figura 7.12: Gráficas de las evoluciones mostradas en la figura 7.1.

Como se observa en la figura 7.1 se obtuvieron 6 cadenas del sistema tag. En la gráfica de frecuencia de la figura 7.12 se tienen el número se observa que los símbolos del alfabeto del sistema tag son los que más fluctúan a través de las evoluciones simuladas, pero estas fluctuaciones son saltos escalonados debido a que durante el proceso permanecen en número hasta que se borran y añaden las cadenas adjuntas. En la gráfica de entropía de la misma figura se observan picos convexos hacia abajo en las evoluciones donde se obtuvieron las cadenas del sistema tag, esto sucede por-

que los símbolos auxiliares del autómata celular aumentan la entropía, pero cuando resultan solo las cadenas del sistema tag el número de símbolos en el espacio disminuye y así mismo la cantidad de información. Esto se observa en las gráficas que presentan en la siguiente figura (Figura 7.13) donde solo se obtuvieron las cadenas del sistema tag sin los pasos intermedios donde se muestra la entropía únicamente de estas cadenas.

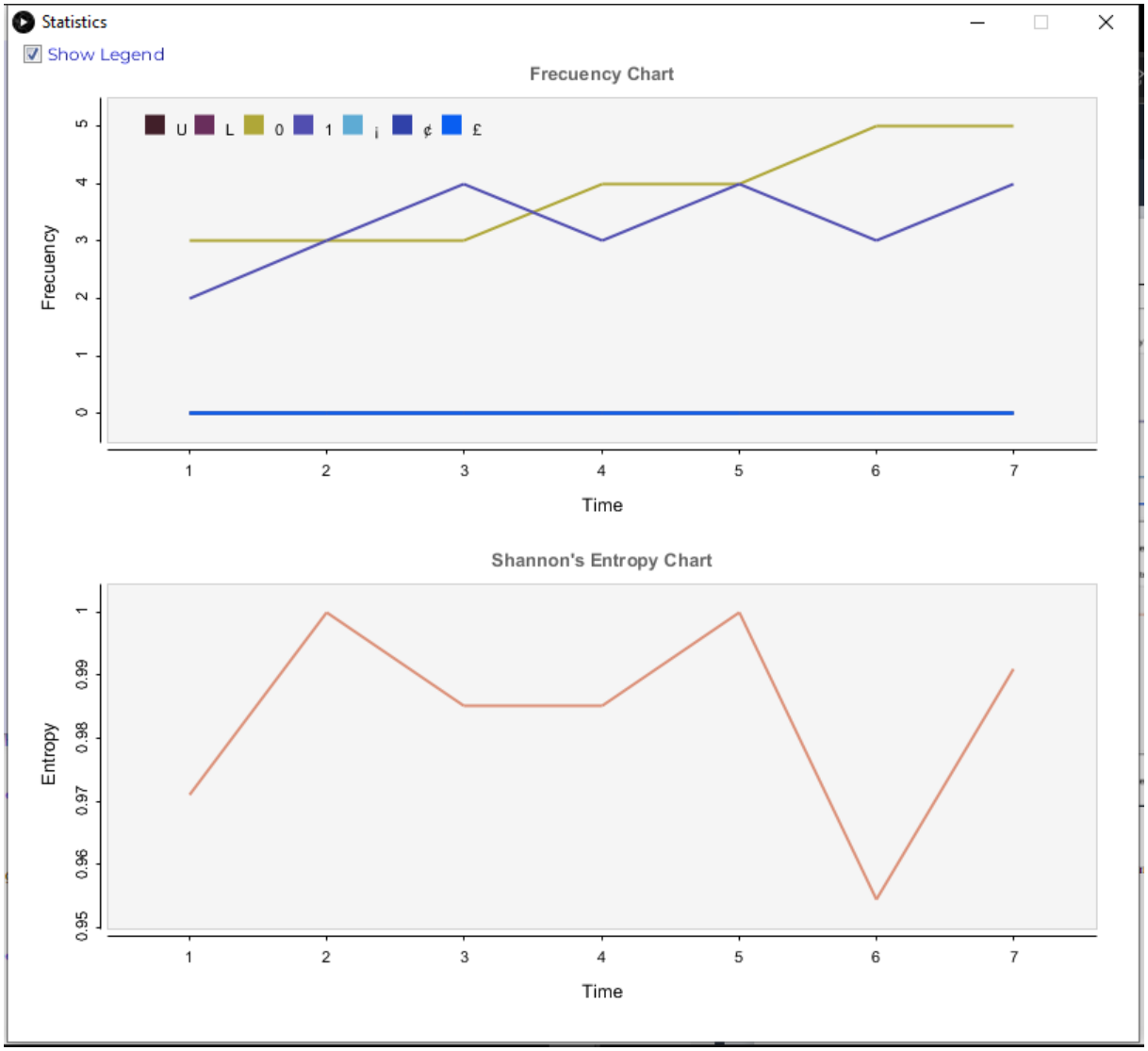


Figura 7.13: Gráficas de las cadenas del sistema tag generadas en la simulación de la figura 7.1.

Otro ejemplo de esto se puede ver en las gráficas de las evoluciones simuladas en la figura 7.7 aunque los picos en la entropía son menos notables en algunas zonas debido a que el sistema tag tiene un mayor número de símbolos que pueden aparecer en el espacio de evoluciones del autómata celular.

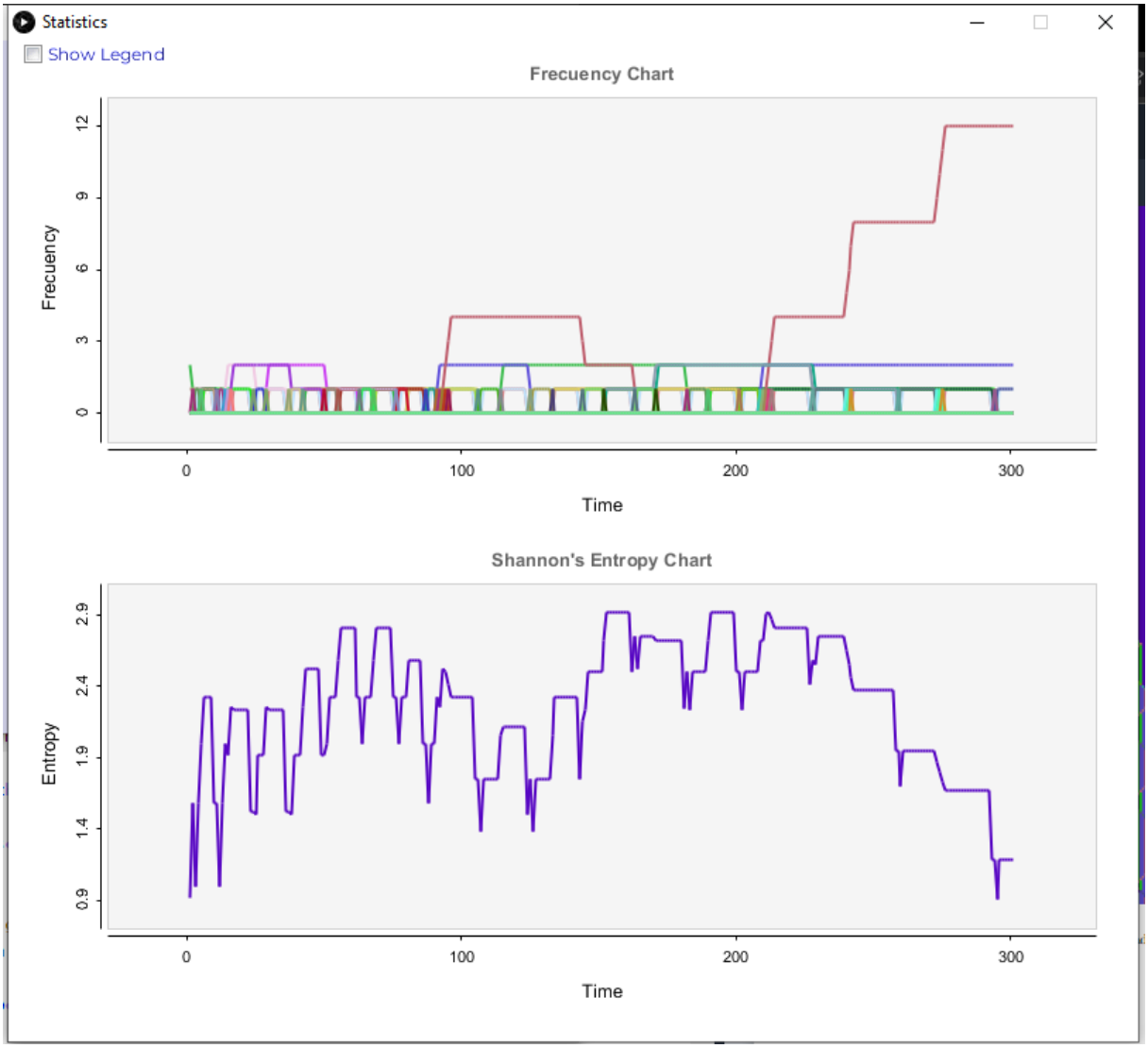


Figura 7.14: Gráficas de las evoluciones mostradas en la figura 7.7.

En las siguientes figuras se muestran las gráficas generadas por las simulaciones de las figuras 7.10 y 7.11 donde se calculan cadenas de sus respectivos sistemas tag en paralelo, en estos no se encuentra un patrón evidente como en las anteriores. Esto

sucede porque no hay una sincronización en el cálculo de cada una de las cadenas, sino que cada cadena se calcula de manera independiente, así que a lo largo de las evoluciones, la frecuencia de cada símbolo es más fluctuante que en el cálculo de una cadena individual.

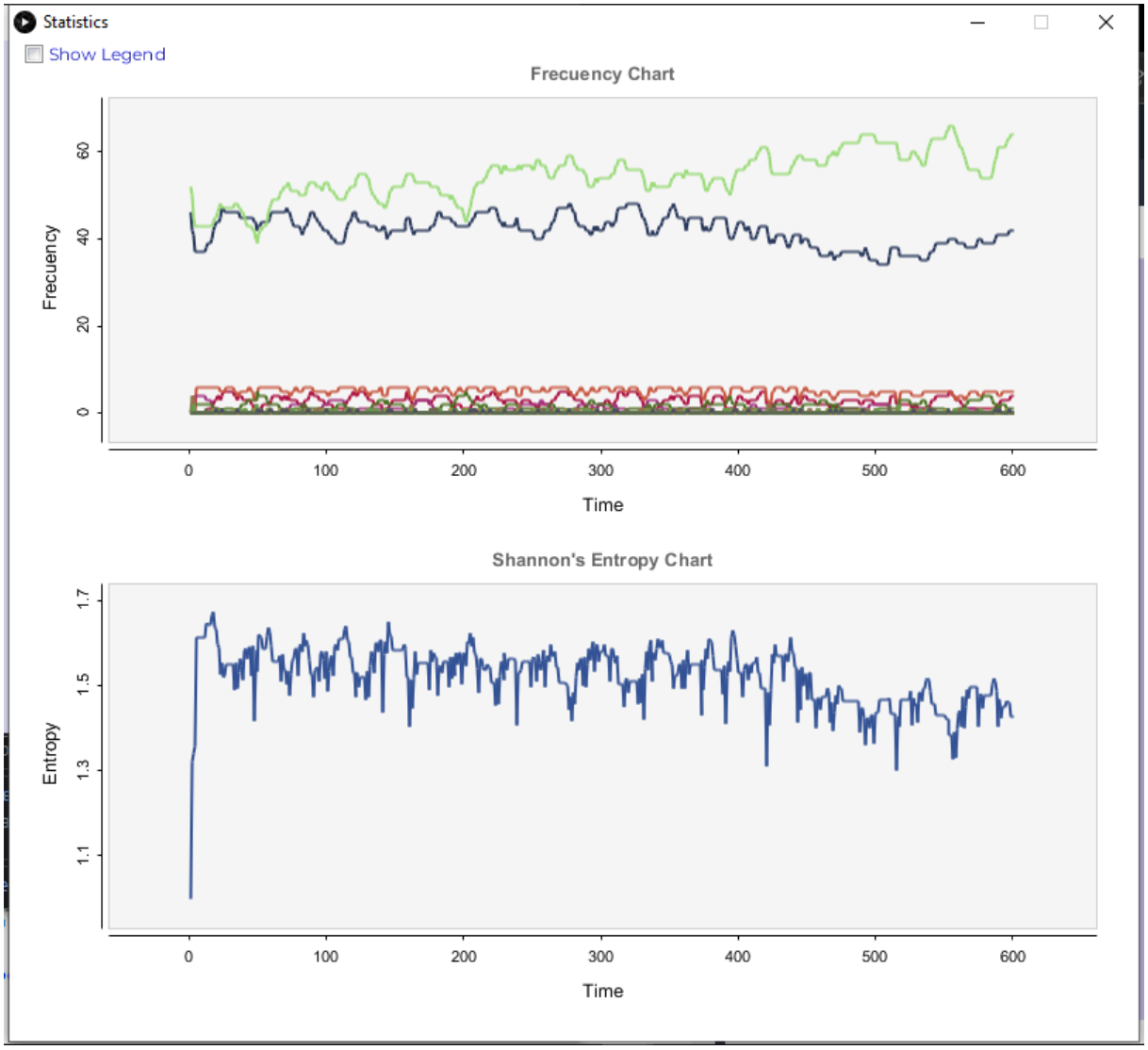


Figura 7.15: Gráficas de las evoluciones mostradas en la figura 7.10.

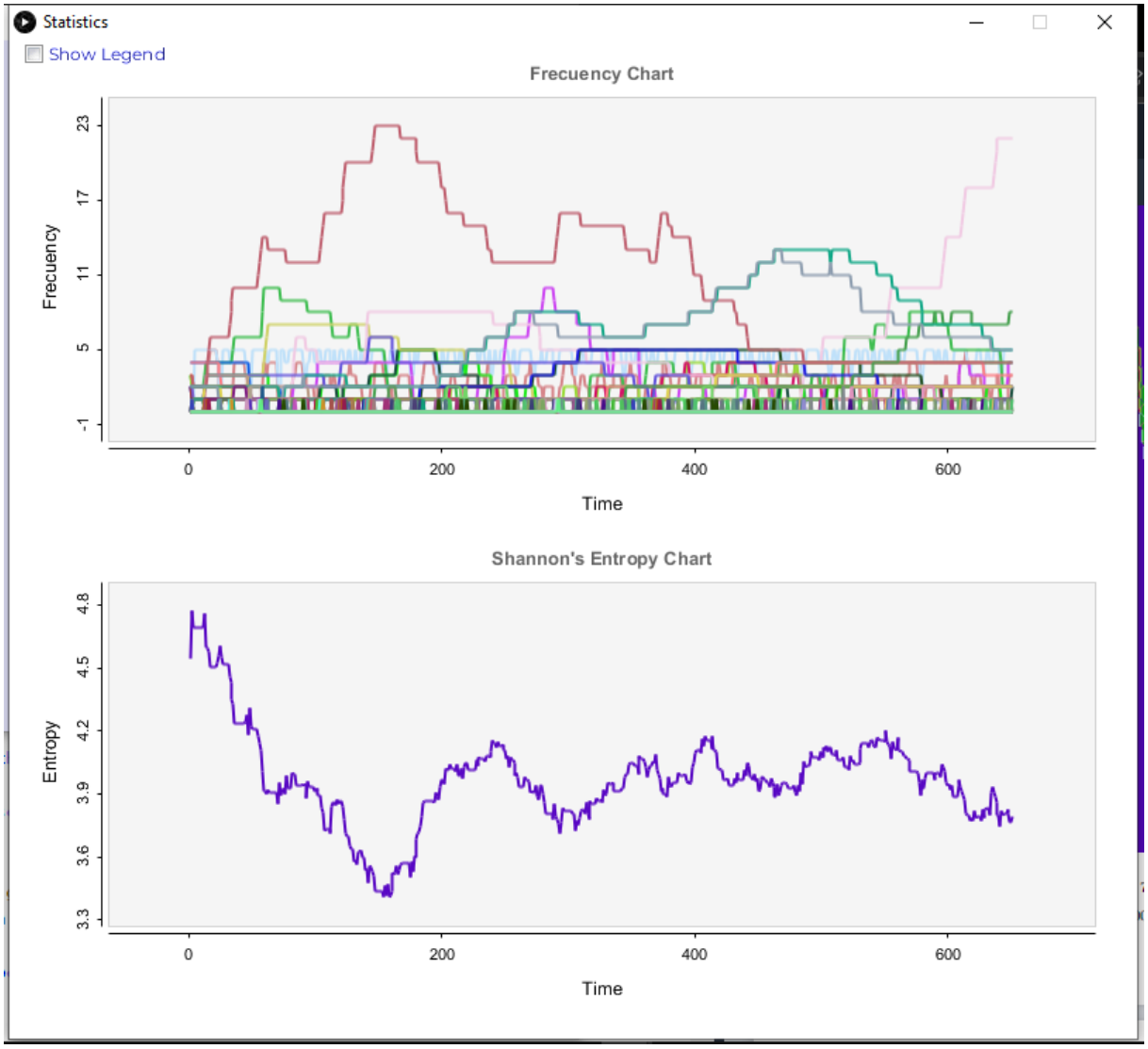


Figura 7.16: Gráficas de las evoluciones mostradas en la figura 7.11.

7.2.3. Computación No Convencional

La computación no convencional trata de quitar el prejuicio que hay sobre el concepto de computadora como un aparato con componentes específicos. Más allá de los componentes físicos, la esencia de la computación es que un mecanismo que muestra un comportamiento complejo se puede construir sin recurrir a componentes cada vez más complejos. Al considerar un conjunto de bloques o elementos, ca-

da uno capaz de calcular una función simple, tal que la salida de cualquiera de estos pueda ser usada como la entrada de otro. Un fenómeno tal como la función de transición de un autómata celular que utiliza los valores de las celdas cercanas a la celda central y junto con esta calcular el valor siguiente.

Las computadoras convencionales usualmente hacen la tarea del procesamiento en serie, es decir, cada tarea se realiza después de otra, pero en la computación no convencional es cada vez más común obtener métodos de paralelización para el cálculo de funciones como se vio en la subsección 7.2.1, esos autómatas celulares son capaces de calcular múltiples cadenas de un solo sistema tag de manera simultánea. Este conjunto de autómatas celulares que simulan sistemas tag pueden ser programados en una máquina de autómatas celulares, esta máquina puede sintetizar diferentes arquitecturas de autómatas celulares de varias dimensiones, por supuesto, los unidimensionales que se trabajaron a lo largo de esta investigación, este enfoque se ha denominado *Materia Programable*, este concepto fue acuñado por Tommaso Toffoli y Norman Margolus en su investigación sobre la creación de un procesador basado en el concepto de autómatas celulares llamado CAM-8 [17] el cual puede procesar millones de celdas y consta de pocos componentes, principalmente una memoria de almacenamiento dinámico y otra de almacenamiento estático. CAM-8 es una malla tridimensional de módulos (similar a un búfer) que lleva a cabo un cálculo paralelo a gran escala. Esta clase de máquinas aún no han sido exploradas en su espectro de computación que es aproximadamente tres órdenes de magnitud mayor a las computadoras convencionales.

Por lo tanto, el conjunto de autómatas celulares que simulan al menos los sistemas tag con número de borrado $P = 2$ que son capaces de computación universal pueden ser programados en arquitecturas de este tipo, siendo así un modelo de computación no convencional basado en autómatas celulares.

Capítulo 8

Conclusiones y Trabajo futuro

8.1. Conclusiones

Al desarrollo del trabajo terminal se obtuvieron respuestas a las preguntas que se habían formulado en el Planteamiento del Problema.

- ¿Qué configuraciones de sistemas tag se pueden convertir en reglas de autómatas celulares?

  Conforme al desarrollo del capítulo 5 se encontró un algoritmo que puede calcular un autómata celular equivalente a un sistema tag arbitrario. Así que cualquier sistema tag con un borrado constante de símbolos y verificación del primer símbolo de la cadena para añadir la cadena adjunta es candidato a tener un autómata celular equivalente. Esto también quedó demostrado mediante la implementación de este algoritmo en el simulador desarrollado en el capítulo 6. Aunque la generación de las cadenas de esta clase de autómatas podría ser optimizado para utilizar menos evoluciones.

- ¿Cómo sería el proceso de conversión?

  El proceso de conversión parece sencillo en la idea general, pero no es una tarea

trivial debido a que el punto más importante de este es establecer las reglas de transición para el autómata celular y como se vio deben ser modeladas muchas reglas para que el autómata celular funcione correctamente. Este proceso está detallado en los ejemplos expuestos en el capítulo 5 donde se explica cómo se generan reglas que pueden resultar en un mecanismo de borrado y concatenación de las cadenas adjuntas del sistema tag mediante de dos tipos de símbolos: auxiliares y de control (banderas). Así mismo se encontró un método heurístico para realizar este proceso explicado en el mismo capítulo.

- ¿Qué características se pueden obtener del análisis de sistemas tag como sistemas dinámicos discretos?

  Durante el desarrollo del capítulo 7 se encontró que un sistema tag se puede analizar en el espacio de los autómatas celulares, algo que les provee de una característica muy notable porque es posible aplicar métodos de análisis de sistemas dinámicos discretos a estos. En el espacio de los autómatas celulares, las cadenas de los sistemas tag presentan un corrimiento a la derecha en un número constante de cadenas igual al número de borrado de estos. Y se necesitan varias evoluciones para calcular cada cadena que sucede a otra.

- ¿Qué características tienen los autómatas celulares que simulan sistemas tag?

  Los autómatas celulares que simulan sistemas tag son capaces de generar todas las cadenas de un sistema tag dada una cadena inicial en su espacio de evoluciones presentando puntos en donde la entropía es proporcional a la de las transiciones del sistema tag. Así mismo son capaces de calcular en paralelo varias cadenas del sistema tag que simulan. En condiciones iniciales aleatorias pueden presentar patrones similares a los generados en el cálculo de cadenas del sistema tag o converger a estados homogéneos, hasta el momento no se han encontrado otra clase de patrones. Pueden ser adaptados en arquitecturas

de computación no convencional basada en autómatas celulares.

Se demostró la factibilidad del estudio debido a que es posible realizar observaciones importantes y destacables sobre el sistema tag, aunque es un sistema sencillo en su funcionamiento, es capaz de realizar tareas muy complejas y es enigmático debido a que no se ha encontrado la manera de optimizar la manera de analizarlo, posiblemente porque no hay una atención especial sobre este, pero está dotado de un buen potencial de computación.

El prototipo de un simulador desarrollado para hacer más eficiente la tarea de analizar la dinámica espacial de los sistemas tag en autómatas celulares sirvió para comprobar una hipótesis que ha surgido desde que se planteó este trabajo terminal: existe un algoritmo tal que dada una definición de un sistema tag cualquiera devuelva un autómata celular equivalente. Cuya respuesta es positiva, aunque no se cataloga como el algoritmo definitivo, sino un indicio de lo que podría ser un método de conversión heurístico.

8.2. Trabajo Futuro

Los objetivos siguientes son propuestos para alcanzar a futuro:

- Implementar más funciones en el simulador tales como simulación de evoluciones ilimitada, exportación en PDF de las gráficas, etc.
- Realizar más conversiones de sistemas tag a autómatas celulares equivalentes para estudiar su dinámica espacial.
- Analizar el comportamiento de más autómatas celulares equivalentes a sistemas tag.

- Optimizar el algoritmo que calcula los autómatas celulares equivalentes para que usen un alfabeto menor y menos evoluciones para el cálculo de cadenas de su sistema tag.
- Crear una librería de las funciones utilizadas en el simulador para que pueda ser usada libremente.

Bibliografía

[1] Cooke J., and Minksy M. (1963). Universality of TAG Systems with P=2. AI Memos 52, MIT, MA.

[2] Cook M. (2004). Universality in Elementary Cellular Automata. Complex Systems, vol. 15. pp. 1–40.

[3] De Mol L. (2008). Tag Systems and Collatz-Like Functions, Theoretical Computer Science, vol. 390, pp. 92–101.

[4] Lothaire, M. (2011). Algebraic Combinatorics on Words, Encyclopedia of Mathematics and Its Applications, 90, Cambridge University Press.

[5] Minsky M. (1961). Recursive Unsolvability of Post's Problem of "Tag."nd other Topics in Theory of Turing Machines. the Annals of Mathematics, 2nd ser., Vol. 74, No. 3.

[6] Neary, Turlough and Woods, Damien. (2009). Four Small Universal Turing Machines. Fundam. Inform.. 91. 123-144. 10.3233/FI-2009-0036.

[7] Neary T., and Cook M. (2018). Generalized Tag Systems. In Reachability Problems, LNCS, vol. 11123, pp. 103–116.

[8] Post E. (1943). Formal reductions of the general combinatorial decision problem. Am. J. Math. 65(2), pp. 197–215.

[9] Von Neumann J. (1966). Theory of Self-Reproducing Automata. University of Illinois Press, Urbana.

[10] Wang H. (1963). Tag systems and lag systems. Math. Ann. 152(4), pp. 65–74.

[11] Wolfram S. (2002). A New Kind of Science. Wolfram Media, Inc.

[12] Wolfram S. (2021). After 100 Years, Can We Finally Crack Post's Problem of Tag? A Story of Computational Irreducibility, and More. Stephen Wolfram Writtings.

[13] Martínez J. Sergio, Mendoza M. Iván, Genaro J. Martínez and Shigeru Ninagawa, Universal One-dimensional Cellular Automata Derived from Turing Machines. IJUC 14.2, p. 121-138.

[14] Reyes M. C. Z., y Peralta R. E. F. (2019). Caracterización de las evoluciones fractales de los autómatas celulares elementales. Escuela Superior de Cómputo, Instituto Politécnico Nacional, Tesis de Ingeniería.

[15] Rogozhin, Yu. (1996). Small Universal Turing Machines, Theoret. Comput. Sci. 168, 215–240.

[16] Russell B., and Whitehead N. A. (1910). Principia Mathematica. Cambridge University Press.

[17] Toffoli, Tommaso, and Margolus, Norman, "Programmable matter," Physica D 47 (1991), 263–272.

[18] Smith A. (2007). Universality of Wolfram's 2, 3 Turing Machine. Complex Systems journal.

Capítulo 9

Glosario

En este último apartado se muestran las definiciones de algunos de los términos o acrónimos presentes en el documento, a fin de facilitar al lector la total comprensión del mismo.

- **Alfabeto.** Es un conjunto finito no vacío, cuyos elementos se denominan letras o símbolos. Denotamos un alfabeto arbitrario con la letra Σ.
- **Complemento Booleano.** Es una operación unitaria cuya aplicación resulta en la negación lógica de un valor. El complemento de 0 es 1 y viceversa.
- **Computación No Convencional.** Es un conjunto de áreas de la ciencia que buscan la implementación de computación basada en el hecho de que la computación es un proceso por el cual los datos son manipulados: adquiriéndolos (entrada), trasformándolos (haciendo cálculos) y transfiríendolos (salida), es decir, cualquier forma de procesamiento ocurrida, ya sea de manera natural o manipulada por algún medio como: computadoras de reacción difusión, algoritmos genéticos, autómatas celulares, computadoras cuánticas, etc.
- **Computación Paralela.** Es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que pro-

blemas grandes, a menudo se pueden dividir en unos más pequeños, que luego son resueltos simultáneamente (en paralelo).

- **Computación Universal.** En la teoría de la computabilidad, se dice que un sistema de reglas de manipulación de datos (como el conjunto de instrucciones de una computadora, un lenguaje de programación o un autómata celular) es Turing completo o computacionalmente universal si se puede usar para simular cualquier máquina de Turing.

- **Diagramas de Atractores.** Es un conjunto de valores hacia los cuales un sistema tiende a evolucionar, dada una gran variedad de condiciones iniciales en el sistema.

- **Diagramas de De Bruijn.** Es una gráfica en donde los nodos son secuencias de símbolos de algún alfabeto y las ligas de los nodos describen como tales secuencias pueden traslapar para formar otras secuencias.

- **Evolución de un autómata celular.** Es una transición que resulta en una configuración global del autómata celular.

- **Expresión regular.** Es una forma de representar los lenguajes regulares, y se construye utilizando caracteres del alfabeto sobre el cual se define el lenguaje.

- **Heurística.** En computación, dos objetivos fundamentales son encontrar algoritmos con buenos tiempos de ejecución y buenas soluciones, usualmente las óptimas. Una heurística es un algoritmo que abandona uno o ambos objetivos; por ejemplo, normalmente encuentran buenas soluciones, aunque no hay pruebas de que la solución no pueda ser arbitrariamente errónea en algunos casos; o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que siempre será así. Las heurísticas generalmente son usadas cuando no

existe una solución óptima bajo las restricciones dadas (tiempo, espacio, etc.), o cuando no existe del todo.

- **Lenguaje de programación.** Es una herramienta que permite desarrollar software o programas para computadora. Los lenguajes de programación son empleados para diseñar e implementar programas encargados de definir y administrar el comportamiento de los dispositivos físicos y lógicos de una computadora.

- **Modelo Computacional.** Es un modelo matemático en las ciencias de la computación que requiere extensos recursos computacionales para estudiar el comportamiento de un sistema complejo por medio de la simulación por computadora.

- **Morfología Matemática.** Es una teoría y técnica para el análisis y tratamiento de las estructuras geométricas, basada en la teoría de conjuntos, teoría de retículos, topología y funciones aleatorias.

- **Patrón fenotípico.** Es un conjunto de rasgos o características observables.

- **Símbolo.** Es una entidad abstracta que no se puede definir, la cual normalmente son letras, dígitos, caracteres. Los símbolos pueden estar formados por varias letras o caracteres.

Capítulo 10

Anexos

En esta sección se encontrarán los anexos del Trabajo Terminal 2020-B100 relacionados con el desarrollo del simulador. El código fuente del prototipo de simulador se encuentra disponible en el siguiente repositorio de GitHub: https://github.com/DerAlanBaruch/gts2ca.

10.1. Clase del Sistema Tag

```
class GTS {
  String[][] rules;
  int p, l;
  String actual;
  color[] cr = {#77E6D9, #E58860};
  String[] e;
  int c = 0;
  int t = 0;
  int tt = 0;
  int m = 0;
  int scl = 1;
  int y = 0;
  StringList caRules = new StringList();
  String[] ect = {"N", "U", "L"};
  StringList eca = new StringList(ect);
  StringList productionRules;
  int randomLen = 10;

  GTS(int _p, String[] _e) {
```

```
20      p = _p;
21      e = _e;
22      l = e.length;
23      rules = new String[p][l];
24      initRules();
25    }
26
27    GTS(int _p) {
28      p = _p;
29    }
30
31    GTS() {
32    }
33
34    void setMode(int _m) {
35      m = _m;
36    }
37
38    void setRandomLen(int _rlen) {
39      randomLen = _rlen;
40    }
41
42    StringList getCARules() {
43      return caRules;
44    }
45
46    StringList getCAAlpha() {
47      return eca;
48    }
49
50    StringList getRules() {
51      return productionRules;
52    }
53
54    String randomInitialString() {
55      String randomInit = "";
56      for (int i = 0; i < randomLen; i++) {
57        randomInit = randomInit + e[int(random(e.length))];
58      }
59      return randomInit;
60    }
61
62    void setRules(StringList _productionRules) {
63      productionRules = _productionRules;
64    }
```

```
65
66  void initRules() {
67    for (int i = 0; i < p; i++) {
68      for (int j = 0; j < l; j++) {
69        rules[i][j] = "";
70      }
71    }
72  }
73
74  void setScale(int _s) {
75    scl = _s;
76  }
77
78  void initColors() {
79    float n = parseFloat(l) - 2.0;
80    cr[0] = color(255, 255, 255);
81    cr[cr.length - 1] = color(0, 0, 0);
82    if (n > 0) {
83      float a = 255.0/(n + 1.0);
84      for (int i = 1; i < cr.length - 1; i++) {
85        cr[i] = color(int(i*a), int(i*a), int(i*a));
86      }
87    }
88  }
89
90  String[] getSymbols() {
91    return e;
92  }
93
94  void setSymbols(String[] _e) {
95    e = _e;
96    l = e.length;
97    rules = new String[p][l];
98    initRules();
99  }
100
101 void setAppendat(int i, int j, String a) {
102   rules[i][j] = a;
103 }
104
105 void printRules() {
106   for (int i = 0; i < p; i++) {
107     printArray(rules[i]);
108   }
109 }
```

```
110
111   void setInitialString(String init) {
112     actual = init;
113   }
114
115   String getActualString() {
116     return actual;
117   }
118
119   int getMode() {
120     return m;
121   }
122
123   void generate() {
124     if (actual.length() >= 1) {
125       String cs = actual.substring(0, 1);
126       if (m == 0) {
127         if (c == 0 && actual.length() >= p) {
128           tt = tt + 1;
129           y = tt;
130           for (int i = 0; i < e.length; i++) {
131             if (e[i].equals(cs)) {
132               actual = actual.substring(p, actual.length()) +
                      ↪  rules[c][i];
133             }
134           }
135         }
136       } else {
137         y = t;
138         for (int i = 0; i < e.length; i++) {
139           if (e[i].equals(cs)) {
140             actual = actual.substring(1, actual.length()) + rules[c][i];
141           }
142         }
143       }
144       t = t + 1;
145       c = t % p;
146     }
147   }
148
149   float entropy() {
150     float n = 0.0;
151     float pr = 0.0;
152     float en = 0.0;
153     for (int j = 0; j < e.length; j++) {
```

```
154       for (int i = 0; i < actual.length(); i++) {
155         if (str(actual.charAt(i)).equals(e[j])) {
156           n = n + 1.0;
157         }
158       }
159       pr = n/parseFloat(actual.length());
160       en = en + log(pow(1/pr, pr))/log(l);
161       n = 0.0;
162     }
163     return en;
164   }
165
166   boolean halt() {
167     if (actual.length() < 1 || y*scl > height) {
168       return true;
169     }
170     return false;
171   }
172
173   int getSteps() {
174     return y;
175   }
176
177   color[] getColors() {
178     return cr;
179   }
180
181   String getRulesString() {
182     String ru = "" + p + "-";
183     for (int i = 0; i < p; i++) {
184       for (int j = 0; j < l; j++) {
185         ru = ru + "(" + i + "," + e[j] +")->" + rules[i][j] + ",";
186       }
187     }
188     return ru;
189   }
190
191   void convertToCA() {
```

```
192     String[] dic = {"¡", "¢", "£", "¤", "¥", "¦", "§", "¨", "©", "ª",
        ↪  "«", "¬", "®", "¯", "°", "±", "²", "³", "´", "µ", "¶", "·", "¸",
        ↪  "¹", "º", "»", "¼", "½", "¾", "¿", "À", "Á", "Â", "Ã", "Ä", "Å",
        ↪  "Æ", "Ç", "È", "É", "Ê", "Ë", "Ì", "Í", "Î", "Ï", "", "Ñ", "Ò",
        ↪  "Ó", "Ô", "Õ", "Ö", "×", "Ø", "Ù", "Ú", "Û", "Ü", "Ý", "", "ß",
        ↪  "á", "â", "ã", "ä", "å", "æ", "ç", "è", "é", "ê", "ë", "ì", "í",
        ↪  "î", "ï", "", "ñ", "ò", "ó", "ô", "õ", "ö", "÷", "ø", "ù", "ú",
        ↪  "û", "ü", "ý", "", "ÿ", "Ā", "ā", "Ă", "ă", "A", "a", "Ć", "ć",
        ↪  "Ĉ", "ĉ", "Ċ", "ċ", "Č", "č", "Ď", "ď", "Đ", "đ", "Ē", "ē", "Ĕ",
        ↪  "Ê", "ê", "Ë", "ë"};
193     StringList ets = new StringList(e);
194     StringList ep = new StringList();
195     StringList es = new StringList();
196     StringList ea = new StringList();
197
198     int ic = 0;
199     int ia = 0;
200     eca.append(ets);
201     for (int i = 0; i < l; i++) {
202       es.appendUnique(dic[ic]);
203       ic = ic + 1;
204     }
205
206     for (int i = 0; i < l*(p - 1); i++) {
207       ep.appendUnique(dic[ic]);
208       ic = ic + 1;
209     }
210
211     for (int i = 0; i< rules[0].length; i++) {
212       for (int j = 0; j< rules[0][i].length() - 1; j++) {
213         ea.appendUnique(dic[ic]);
214         ic = ic + 1;
215       }
216     }
217
218     eca.append(es);
219     eca.append(ep);
220     eca.append(ea);
221
222     if (ep.size() > 0)
223       caRules.append("N[" + join(ep.array(), "") + "].{1}->N");
224     caRules.append(".{1}N[" + join(es.array(), "") + "]->L");
225     caRules.append(".{1}L[" + join(ets.array(), "") + "]->L");
226     caRules.append("NN[NL" + join(ets.array(), "") + join(ep.array(), "")
        ↪  + "]->N");
```

```
227     caRules.append("[U" + join(ets.array(), "") + join(ep.array(), "") +
        ↪  "]N[NL]->N");
228     caRules.append("[L" + join(ets.array(), "") + "][" +
        ↪  join(ets.array(), "") + "]U->U");
229     caRules.append("LU[" + join(ets.array(), "") + "]->N");
230     caRules.append(".{1}LU->N");
231
232     for (int i = 0; i < ets.size(); i++) {
233       caRules.append("[LU" + join(ets.array(), "") + "]" + ets.get(i) +
          ↪  "[NL" + join(ets.array(), "") + join(es.array(), "") +
          ↪  join(ea.array(), "") + "]->" + ets.get(i));
234       caRules.append(".{1}[" + join(es.array(), "") + "]" + ets.get(i) +
          ↪  "->" + ets.get(i));
235       caRules.append(ets.get(i) + "U.{1}->" + ets.get(i));
236       if (ep.size() > 0)
237         caRules.append("N" + ets.get(i) + ".{1}->" + ep.get(i));
238       else
239         caRules.append("N" + ets.get(i) + ".{1}->" + es.get(i));
240     }
241
242     StringList eps = new StringList(ep);
243     eps.append(es);
244     for (int i = 0; i < ep.size(); i++) {
245       caRules.append(eps.get(i) + ".{2}->" + eps.get(i + 1));
246     }
247     for (int i = 0; i < es.size(); i++) {
248       caRules.append(es.get(i) + "[" + join(ets.array(), "") + "].{1}->"
          ↪  + es.get(i));
249     }
250
251     int k = 0;
252     for (int i = 0; i< rules[0].length; i++) {
253       if (rules[0][i].length() > 0) {
254         for (int j = 0; j < rules[0][i].length(); j++) {
255           if (j == 0) {
256             if (rules[0][i].length() == 1) caRules.append(es.get(i) +
                ↪  "[NL].{1}" + "->U");
257             else caRules.append(es.get(i) + "[NL].{1}" + "->" +
                ↪  ea.get(k));
258             caRules.append("[N" + join(ets.array(), "") + "]" + es.get(i)
                ↪  + "[NL]" + "->" + rules[0][i].charAt(j));
259           } else {
260             if (j == rules[0][i].length() - 1) caRules.append(ea.get(k) +
                ↪  "[NL].{1}" + "->U");
```

```
261             else caRules.append(ea.get(k) + "[NL].{1}" + "->" + ea.get(k
                ↪  + 1));
262             caRules.append("[N" + join(ets.array(), "") + "]" + ea.get(k)
                ↪  + "[NL]" + "->" + rules[0][i].charAt(j));
263             k++;
264           }
265         }
266       } else {
267         caRules.append(es.get(i) + "[NL].{1}" + "->N");
268         caRules.append("[N" + join(ets.array(), "") + "]" + es.get(i) +
            ↪  "[NL]" + "->U");
269       }
270     }
271
272     println(eca);
273     println(caRules);
274   }
275 }
```

10.2. Clase del Autómata Celular

```
1  import java.util.Map;
2  import grafica.*;
3
4  class CA {
5    ArrayList<GPointsArray> pointsFrecuency = new
     ↪  ArrayList<GPointsArray>();
6    GPointsArray pointsEntropy = new GPointsArray();
7    PGraphics pg;
8    String[] cells;
9    int generation = 0;
10   int scl = 1;
11   String init;
12   String[] e;
13   color[] ce;
14   StringList all = new StringList();
15   StringList allTS = new StringList();
16   int h;
17   int evolutions = 0;
18   boolean showStroke = false;
19   String imgPath = "";
20   String[] ets;
21   StringList rules = new StringList();
22   ArrayList<String[]> rulesList = new ArrayList<String[]>();
```

```
23  color colorStroke = color(32);
24  HashMap<String, Integer> sigmaColor = new HashMap<String, Integer>();
25  boolean onlyTS = false;
26
27  CA() {
28  }
29
30  CA(StringList _rules, StringList alpha) {
31    e = alpha.array();
32    rules = _rules;
33    for (int i = 0; i < rules.size(); i++) {
34      String[] ru = rules.get(i).split("->");
35      rulesList.add(ru);
36    }
37  }
38
39  void setETS(String[] _ets) {
40    ets = _ets;
41  }
42
43  void setPGraphics(PGraphics _pg) {
44    pg = _pg;
45  }
46
47  void setOnlyTS(boolean _ots) {
48    onlyTS = _ots;
49  }
50
51  String[] getAlpha() {
52    return e;
53  }
54
55  color[] getAlphaColors() {
56    return ce;
57  }
58
59  void setAlphaColors(color[] _ce) {
60    ce = _ce;
61    makeSigmaColorHash();
62  }
63
64  String executeRules (String strp, String a) {
65    for (String[] rule : rulesList) {
66      if (match(strp, rule[0]) != null) {
67        return rule[1];
```

```
68        }
69      }
70      return a;
71    }
72
73    String randomInitialString() {
74      String randomInit = "";
75      for (int i = 0; i < cells.length; i++) {
76        randomInit = randomInit + e[int(random(e.length))];
77      }
78      return randomInit;
79    }
80
81    void initialize(String _init, int _w, int _h) {
82      all = new StringList();
83      all.clear();
84      allTS = new StringList();
85      allTS.clear();
86      cells = new String[_w];
87      init = _init;
88      generation = 0;
89      evolutions = 0;
90      h = _h;
91      if (cells.length > init.length()) {
92        cells[0] = "N";
93        for (int i = 1; i < init.length() + 1; i++) {
94          cells[i] = str(init.charAt(i - 1));
95        }
96        for (int i = init.length() + 1; i < cells.length; i++) {
97          cells[i] = "N";
98        }
99      } else {
100       for (int i = 0; i < cells.length; i++) {
101         cells[i] = str(init.charAt(i));
102       }
103     }
104   }
105
106   void generatePoints() {
107     pointsFrecuency = new ArrayList<GPointsArray>();
108     pointsEntropy = new GPointsArray();
109     for (int i = 0; i < e.length; i++) {
110       pointsFrecuency.add(new GPointsArray());
111     }
112
```

```
113     StringList toAll = new StringList();
114     if (onlyTS) toAll = allTS;
115     else toAll = all;
116     int allSize = toAll.size();
117     for (int j = 0; j < allSize; j++) {
118       StringList cArray = new StringList(split(toAll.get(j), ","));
119       IntDict tally = cArray.getTally();
120       float h = 0.0;
121       for (int k = 0; k < e.length; k++) {
122         GPointsArray gpf = pointsFrecuency.get(k);
123         if (tally.hasKey(e[k])) {
124           gpf.add(j + 1, tally.get(e[k]));
125           if (!e[k].equals("N")) {
126             float p;
127             if (tally.hasKey("N"))
128               p = float(tally.get(e[k]))/float(cells.length -
                  ↪ tally.get("N"));
129             else
130               p = float(tally.get(e[k]))/float(cells.length);
131             h = h - p*log(p)/log(2);
132           }
133         } else {
134           gpf.add(j + 1, 0);
135         }
136       }
137       pointsEntropy.add(j + 1, h);
138     }
139   }
140
141   ArrayList<GPointsArray> getGPointsFrecuency() {
142     return pointsFrecuency;
143   }
144
145   GPointsArray getGPointsEntropy() {
146     return pointsEntropy;
147   }
148
149   void generate() {
150     String stringcells = join(cells, ",");
151     evolutions++;
152     if (onlyTS) {
153       if (match(join(cells, ""), "[N" + join(ets, "") +
          ↪ "]{"+cells.length+"}") != null) {
154         allTS.append(stringcells);
155         generation++;
```

```
156         println(evolutions);
157       }
158     } else {
159       all.append(stringcells);
160       generation++;
161     }
162     String[] nextgen = new String[cells.length];
163     //println(join(cells, "&")+"\\\\\\hline");
164     for (int i = 0; i < cells.length; i++) {
165       String left = cells[(cells.length + i - 1) % cells.length];
166       String right = cells[(i + 1) % cells.length];
167       nextgen[i] = executeRules(left + cells[i] + right, left);
168     }
169     for (int i = 0; i < cells.length; i++) {
170       cells[i] = nextgen[i];
171     }
172   }
173
174   PGraphics getPG() {
175     return pg;
176   }
177
178   int getEvolutions() {
179     return evolutions;
180   }
181
182   void setScale(int _scl) {
183     scl = _scl;
184   }
185
186   void setShowStroke(boolean _show) {
187     showStroke = _show;
188   }
189
190   void setImgPath(String _path) {
191     imgPath = _path;
192   }
193
194   void saveEvolutions() {
195     saveStrings("frame-" + frameCount + ".txt", all.array());
196   }
197
198   void setColorStroke(color _cs) {
199     colorStroke = _cs;
200   }
```

```
201
202   color getColorStroke() {
203     return colorStroke;
204   }
205
206   void makeSigmaColorHash() {
207     for (int j = 0; j < e.length; j++) {
208       sigmaColor.put(e[j], ce[j]);
209     }
210   }
211
212   StringList getAllStrings() {
213     StringList toAll = new StringList();
214     if (onlyTS) toAll = allTS;
215     else toAll = all;
216     return toAll;
217   }
218
219   PGraphics render() {
220     int ge = 0;
221     StringList toAll = new StringList();
222     if (onlyTS) toAll = allTS;
223     else toAll = all;
224     int allSize = toAll.size();
225     pg = createGraphics(cells.length*scl, h*scl);
226     pg.beginDraw();
227     for (int j = 0; j < allSize; j++) {
228       String[] cArray = split(toAll.get(j), ",");
229       for (int i = 0; i < cArray.length; i++) {
230         if (scl > 1) {
231           pg.fill(sigmaColor.get(cArray[i]));
232           if (showStroke)
233             pg.stroke(colorStroke);
234           else pg.noStroke();
235           pg.rect(i*scl, ge*scl, scl, scl);
236         } else {
237           pg.set(i, ge, sigmaColor.get(cArray[i]));
238         }
239       }
240       ge++;
241     }
242     pg.endDraw();
243     return pg;
244   }
245
```

```
246   void reset() {
247     pointsFrecuency = new ArrayList<GPointsArray>();
248     pointsEntropy = new GPointsArray();
249     pg = null;
250     generation = 0;
251     all = new StringList();
252   }
253 
254   boolean finished() {
255     if (generation > h) {
256       return true;
257     } else {
258       return false;
259     }
260   }
261 }
```

10.3. Código de la Interfaz Gráfica

```
1  import g4p_controls.*;
2  import java.util.*;
3  import grafica.*;
4  GTS gts;
5  CA ca = null;
6  boolean onetime = false;
7  boolean onetimePlot = true;
8  float posX = 0;
9  float posY = 0;
10 float dzoomX = 0;
11 float dzoomY = 0;
12 boolean toDrag = false;
13 PGraphics pg, pg1, pg12;
14 float zoom = 0;
15 float nh = 1400;
16 float h = nh;
17 float nw = 350;
18 float w = nw;
19 boolean isTS = false;
20 String[] alpha;
21 color[] alphaColors;
22 int selectedColor = 0;
23 PGraphics pgSimulator;
24 int sclValue = 1;
25 GPlot plotFrecuency;
```

```
26 GPlot plotEntropy;
27 GPointsArray gpe;
28 ArrayList<GPointsArray> gpaList = new ArrayList<GPointsArray>();
29
30 java.awt.Font bold16 = new java.awt.Font("Montserrat",
   ↪  java.awt.Font.BOLD, 16);
31 java.awt.Font bold12 = new java.awt.Font("Montserrat",
   ↪  java.awt.Font.BOLD, 12);
32 java.awt.Font plain16 = new java.awt.Font("Montserrat",
   ↪  java.awt.Font.PLAIN, 16);
33 java.awt.Font plain12 = new java.awt.Font("Montserrat",
   ↪  java.awt.Font.PLAIN, 12);
34
35 void settings() {
36   size(280, 490, JAVA2D);
37 }
38
39 GWindow winNewTS = null, winSimulatorCA = null, winColors = null,
   ↪  winStatistics = null, winTSLengthRandom = null;
40 GTextArea txa1;
41 GTextField txf1, txf2, txf3, txfw, txfh;
42 GSlider sdr1;
43 GToggleGroup togG1Options, togGSptions;
44 GOption grp1_a, grp1_b, grp1_c, grp1_d;
45 GDropList cdl;
46 GLabel label3color, label1principal;
47 GView viewSimulator;
48 GButton btnSaveImgPNG, btnSaveImgTIFF, btnSaveImgTXT, btnSaveImgJPG;
49 GPanel filePrincipal, saveImg, zoomPanel;
50 GCheckbox cbx1, cbx2;
51
52 void setup() {
53   createGUI();
54 }
55
56 void draw() {
57   background(230, 230, 255);
58   noFill();
59   strokeWeight(2);
60   stroke(197, 206, 232);
61   rect(10, 30, 260, 80);
62   rect(10, 120, 260, 360);
63 }
64
65 void win_newts_draw(PApplet appc, GWinData data) {
```

```
66    appc.background(230, 230, 255);
67  }
68
69  void win_colors_draw(PApplet appc, GWinData data) {
70    appc.background(230, 230, 255);
71  }
72
73  void generateNext() {
74    while (onetime && !ca.finished()) {
75      ca.generate();
76      ca.generatePoints();
77    }
78    onetime = false;
79  }
80
81  void win_simulatorca_draw(PApplet appc, GWinData data) {
82    appc.background(255);
83    if (!ca.finished()) {
84      pg = ca.render();
85    }
86    appc.image(pg, posX, posY, w*(1+zoom), h*(1+zoom));
87  }
88
89  void win_statistics_draw(PApplet appc, GWinData data) {
90    appc.background(255);
91    appc.frameRate(5);
92    float[] legendX = new float[alpha.length - 1];
93    float[] legendY = new float[alpha.length - 1];
94    int lY = 0;
95    for (int i = 0; i < legendX.length; i++) {
96      legendX[i] = 0.07 + 0.05*(i % 20);
97      legendY[i] = 0.92 - 0.05*lY;
98      if (i % 20 == 19) lY++;
99    }
100   gpaList = ca.getGPointsFrecuency();
101   gpe = ca.getGPointsEntropy();
102   if (onetimePlot) {
103     plotFrecuency = new GPlot(appc);
104     plotFrecuency.setPos(0, 0);
105     plotFrecuency.setOuterDim(800, (displayHeight - 80) / 2);
106     plotEntropy = new GPlot(appc);
107     plotEntropy.setPos(0, (displayHeight - 80) / 2);
108     plotEntropy.setOuterDim(800, (displayHeight - 80) / 2);
109     plotFrecuency.setTitleText("Frecuency Chart");
110     plotFrecuency.getXAxis().setAxisLabelText("Time");
```

```
111       plotFrecuency.getYAxis().setAxisLabelText("Frecuency");
112       plotEntropy.setTitleText("Shannon's Entropy Chart");
113       plotEntropy.getXAxis().setAxisLabelText("Time");
114       plotEntropy.getYAxis().setAxisLabelText("Entropy");
115       plotFrecuency.activatePanning();
116       plotFrecuency.activateZooming(1.2f, CENTER, CENTER);
117       plotEntropy.activatePanning();
118       plotEntropy.activateZooming(1.2f, CENTER, CENTER);
119
120       plotEntropy.setPoints(gpe);
121       plotEntropy.setLineColor(alphaColors[0]);
122       plotEntropy.setLineWidth(2);
123       plotFrecuency.setPoints(gpaList.get(1));
124       plotFrecuency.setLineColor(alphaColors[1]);
125       plotFrecuency.setLineWidth(2);
126       for (int i = 2; i < gpaList.size(); i++) {
127         plotFrecuency.addLayer(alpha[i], gpaList.get(i));
128         plotFrecuency.getLayer(alpha[i]).setLineColor(alphaColors[i]);
129         plotFrecuency.getLayer(alpha[i]).setLineWidth(2);
130       }
131       onetimePlot = false;
132     }
133     plotEntropy.setPoints(gpe);
134     plotFrecuency.setPoints(gpaList.get(1));
135     for (int i = 2; i < gpaList.size(); i++) {
136       plotFrecuency.getLayer(alpha[i]).setPoints(gpaList.get(i));
137     }
138     plotFrecuency.beginDraw();
139     plotFrecuency.drawBox();
140     plotFrecuency.drawXAxis();
141     plotFrecuency.drawYAxis();
142     plotFrecuency.drawTitle();
143     plotFrecuency.drawLines();
144     if (cbx2.isSelected())
145       plotFrecuency.drawLegend((new
          ↪  StringList(alpha)).getSubset(1).array(), legendX, legendY);
146     plotFrecuency.endDraw();
147     plotEntropy.beginDraw();
148     plotEntropy.drawBox();
149     plotEntropy.drawXAxis();
150     plotEntropy.drawYAxis();
151     plotEntropy.drawTitle();
152     plotEntropy.drawLines();
153     plotEntropy.endDraw();
154   }
```

```
155
156 public void SimulatorCAMouse(PApplet appc, GWinData data, MouseEvent
    ↪ event) {
157   switch(event.getAction()) {
158   case MouseEvent.PRESS:
159     if (!toDrag) {
160       dzoomX = appc.mouseX - posX;
161       dzoomY = appc.mouseY - posY;
162       toDrag = true;
163     }
164     break;
165   case MouseEvent.RELEASE:
166     if (toDrag) {
167       toDrag = false;
168     }
169     break;
170   case MouseEvent.WHEEL:
171     float e = event.getCount();
172     if (e > 0 && float(nf(zoom, 0, 1)) < 2) {
173       zoom = zoom + 0.1;
174     }
175     if (e < 0 && float(nf(zoom, 0, 1)) > -0.5) {
176       zoom = zoom - 0.1;
177     }
178     if (posX + w*(1+zoom) < 1.0)
179       posX = appc.width - w*(1+zoom);
180     if (posY + h*(1+zoom) < 1.0)
181       posY = appc.height - h*(1+zoom);
182     break;
183   case MouseEvent.DRAG:
184     if ((appc.mouseY - dzoomY) < 1.0 && (appc.mouseY - dzoomY +
      ↪ h*(1+zoom)) > appc.height) posY = appc.mouseY - dzoomY;
185     else {
186       if ((appc.mouseY - dzoomY) > 1.0)
187         posY = 0.0;
188       else if (h*(1+zoom) >= appc.height)
189         posY = appc.height - h*(1+zoom);
190     }
191     if ((appc.mouseX - dzoomX) < 1.0 && (appc.mouseX - dzoomX +
      ↪ w*(1+zoom)) > appc.width) posX = appc.mouseX - dzoomX;
192     else {
193       if ((appc.mouseX - dzoomX) > 1.0)
194         posX = 0.0;
195       else if (w*(1+zoom) >= appc.width)
196         posX = appc.width - w*(1+zoom);
```

```
197       }
198       break;
199     }
200   }
201
202   void btnNewTSCallback(GButton source, GEvent event) {
203     if (winNewTS == null) {
204       createNewTSWindow();
205     } else {
206       winNewTS.setVisible(!winNewTS.isVisible());
207       winNewTS = null;
208     }
209   }
210
211   void btnSaveImgCallback(GButton source, GEvent event) {
212     String ext = "";
213     if (source == btnSaveImgPNG) ext = ".png";
214     if (source == btnSaveImgTIFF) ext = ".tiff";
215     if (source == btnSaveImgJPG) ext = ".jpg";
216     String imgPath = G4P.selectOutput("Save image as...", ext, "Image
        ↪ files");
217     if (ca != null && imgPath != null) {
218       if (imgPath.trim().toLowerCase().endsWith(ext))
219         ca.render().save(imgPath);
220       else
221         ca.render().save(imgPath + ext);
222     }
223     saveImg.setCollapsed(true);
224   }
225
226   void btnSaveTextCallback(GButton source, GEvent event) {
227     String imgPath = G4P.selectOutput("Save text as...", ".csv", "CSV
        ↪ files");
228     if (ca != null && imgPath != null) {
229       if (imgPath.trim().toLowerCase().endsWith(".csv"))
230         saveStrings(imgPath, ca.getAllStrings().array());
231       else
232         saveStrings(imgPath + ".csv", ca.getAllStrings().array());
233     }
234     saveImg.setCollapsed(true);
235   }
236
237   void btnLengthRandomCallback(GButton source, GEvent event) {
238     if (winTSLengthRandom == null) {
239       createLengthRandomWindow();
```

```
240   } else {
241     winTSLengthRandom.close();
242     winTSLengthRandom = null;
243   }
244 }
245 
246 void btnSimulatorCACallback(GButton source, GEvent event) {
247   if (winSimulatorCA == null) {
248     createSimulatorCAWindow();
249   } else {
250     winSimulatorCA.close();
251   }
252 }
253 
254 void closeSimulatorCACallback(GWindow window) {
255   winSimulatorCA = null;
256   onetime = false;
257   println("Simulator closed");
258 }
259 
260 void closeStatisticsCallback(GWindow window) {
261   winStatistics = null;
262   onetimePlot = true;
263   println("Statistics closed");
264 }
265 
266 void closeRandomTSCallback(GWindow window) {
267   winTSLengthRandom = null;
268 }
269 
270 void btnStatisticsCallback(GButton source, GEvent event) {
271   if (winStatistics == null) {
272     createStatisticsWindow();
273   } else {
274     winStatistics.close();
275   }
276 }
277 
278 void btnColorsCallback(GButton source, GEvent event) {
279   if (winColors == null) {
280     createColorsWindow();
281   } else {
282     winColors.setVisible(!winColors.isVisible());
283     winColors = null;
284   }
```

```
285 }
286
287 GTS makeTagSystem(String[] eb, int p, String[] rs) {
288   printArray(eb);
289   printArray(rs);
290   GTS gts = new GTS(p, eb);
291   for (int i = 0; i < eb.length; i++) {
292     gts.setAppendat(0, i, rs[i]);
293   }
294   gts.convertToCA();
295   println(gts.getRulesString());
296   return gts;
297 }
298
299 void createTS(GButton source, GEvent event) {
300   String prodRulesStr = txa1.getText().replace(" ", "");
301   println(prodRulesStr);
302   int p = int(txf1.getText().trim());
303   println(p);
304   if (prodRulesStr != "" && p != 0) {
305     String[] rs = prodRulesStr.split(",");
306     StringList sigma = new StringList();
307     StringList appendants = new StringList();
308     StringList prodRulesList = new StringList();
309     for (int i = 0; i < rs.length; i++) {
310       prodRulesList.append(rs[i].trim());
311       String[] rn = rs[i].split("->");
312       sigma.append(rn[0].trim());
313       if (rn.length > 1)
314         appendants.append(rn[1].trim());
315       else
316         appendants.append("");
317     }
318     pg = null;
319     ca = null;
320     gts = makeTagSystem(sigma.array(), p, appendants.array());
321     gts.setRules(prodRulesList);
322     ca = new CA(gts.getCARules(), gts.getCAAlpha());
323     String tsMsg = "TS: R=" + appendants.array().length + " P=" + p;
324     String caMsg = "CA: R=" + gts.getCARules().size() + " Sigma=" +
      ↪  gts.getCAAlpha().size();
325     StringBuilder s = new StringBuilder();
326     s.append(tsMsg + "\n");
327     s.append(caMsg);
328     label1principal.setText(s.toString());
```

```
329     println(s.toString());
330     label1principal.setTextBold();
331     ca.setETS(gts.getSymbols());
332     alpha = ca.getAlpha();
333     float diff = 1.0 / float(alpha.length);
334     IntList alphaColorList = new IntList();
335     for (int i = 0; i < alpha.length; i++) {
336       alphaColorList.append(lerpColor(color(int(random(255)),
          ↪ int(random(255)), int(random(255))), color(int(random(255)),
          ↪ int(random(255)), int(random(255))), i*diff));
337     }
338     alphaColors = alphaColorList.array();
339     ca.setAlphaColors(alphaColorList.array());
340     isTS = true;
341     winNewTS.setVisible(false);
342     winNewTS = null;
343   } else {
344     G4P.showMessage(this, "Please, enter the production rules and
       ↪ deletion number P", "Data missed", G4P.ERROR_MESSAGE);
345   }
346 }
347
348 void createNewTSWindow() {
349   winNewTS = GWindow.getWindow(this, "New", 50, 50, 300, 290, JAVA2D);
350   winNewTS.setActionOnClose(G4P.HIDE_WINDOW);
351   winNewTS.addDrawHandler(this, "win_newts_draw");
352   GLabel label1 = new GLabel(winNewTS, 10, 10, 280, 40);
353   label1.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
354   label1.setText("Enter the production rules and deletion number P, then
     ↪ clic on Create.");
355   label1.setOpaque(false);
356   label1.setFont(new java.awt.Font("Montserrat", java.awt.Font.PLAIN,
     ↪ 12));
357   txa1 = new GTextArea(winNewTS, 10, 60, 280, 160);
358   txa1.setPromptText("Production rules* (ej. 0->01, 1->b0)");
359   txa1.setText("");
360   txf1 = new GTextField(winNewTS, 10, 230, 140, 20);
361   txf1.setPromptText("Deletion number P*");
362   txf1.setText("");
363   GButton summit = new GButton(winNewTS, 200, 260, 90, 20);
364   summit.setText("Create");
365   summit.addEventHandler(this, "createTS");
366   filePrincipal.setCollapsed(true);
367 }
368
```

```
369 void createLengthRandomWindow() {
370   if (isTS) {
371     winTSLengthRandom = GWindow.getWindow(this, "Length of String", 50,
      ↪  50, 200, 120, JAVA2D);
372     winTSLengthRandom.setActionOnClose(G4P.CLOSE_WINDOW);
373     winTSLengthRandom.addDrawHandler(this, "win_newts_draw");
374     winTSLengthRandom.addOnCloseHandler(this, "closeRandomTSCallback");
375     GLabel label1 = new GLabel(winTSLengthRandom, 10, 10, 180, 40);
376     label1.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
377     label1.setText("Enter a length for the random TS string:");
378     label1.setOpaque(false);
379     label1.setFont(new java.awt.Font("Montserrat", java.awt.Font.PLAIN,
      ↪  12));
380     txf3 = new GTextField(winTSLengthRandom, 10, 60, 180, 20);
381     txf3.setPromptText("Length*");
382     txf3.setText("");
383     GButton summit = new GButton(winTSLengthRandom, 100, 90, 90, 20);
384     summit.setText("OK");
385     summit.addEventHandler(this, "generateRandomStringTS");
386   } else {
387     G4P.showMessage(this, "There is not a cellular automata, create new
      ↪  or open some one in File", "Cellular automata missed",
      ↪  G4P.ERROR_MESSAGE);
388   }
389 }
390
391 public void handleColorChooser(GButton button, GEvent event) {
392   alphaColors[selectedColor] = G4P.selectColor();
393   ca.setAlphaColors(alphaColors);
394   pg = ca.render();
395   pg1.beginDraw();
396   pg1.background(alphaColors[selectedColor]);
397   pg1.endDraw();
398 }
399
400 public void handleColorChooserStroke(GButton button, GEvent event) {
401   color colorStroke = G4P.selectColor();
402   ca.setColorStroke(colorStroke);
403   pg = ca.render();
404   pg12.beginDraw();
405   pg12.background(colorStroke);
406   pg12.endDraw();
407 }
408
409 void btnColorsRandomizeCallback(GButton source, GEvent event) {
```

```
410   float diff = 1.0 / float(alpha.length);
411   IntList alphaColorList = new IntList();
412   for (int i = 0; i < alpha.length; i++) {
413     alphaColorList.append(lerpColor(color(int(random(255)),
        ↪ int(random(255)), int(random(255))), color(int(random(255)),
        ↪ int(random(255)), int(random(255))), i*diff));
414   }
415   alphaColors = alphaColorList.array();
416   ca.setAlphaColors(alphaColorList.array());
417   pg1.beginDraw();
418   pg1.background(alphaColors[selectedColor]);
419   pg1.endDraw();
420   pg = ca.render();
421 }
422
423 void btnColorsOkCallback(GButton source, GEvent event) {
424   winColors.close();
425   winColors = null;
426 }
427
428 void alphaChooser(GDropList droplist, GEvent event) {
429   selectedColor = cdl.getSelectedIndex();
430   label3color.setText(alpha[selectedColor]);
431   label3color.setTextBold();
432   pg1.beginDraw();
433   pg1.background(alphaColors[selectedColor]);
434   pg1.endDraw();
435 }
436
437 void createColorsWindow() {
438   if (isTS) {
439     winColors = GWindow.getWindow(this, "Colors", 50, 50, 240, 250,
        ↪ JAVA2D);
440     winColors.setActionOnClose(G4P.CLOSE_WINDOW);
441     winColors.addDrawHandler(this, "win_colors_draw");
442     GLabel label1 = new GLabel(winColors, 10, 10, 220, 40);
443     label1.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
444     label1.setText("Select a symbol and clic in Change to change its
        ↪ color.");
445     label1.setOpaque(false);
446     label1.setFont(plain12);
447     GLabel label2 = new GLabel(winColors, 10, 50, 220, 20);
448     label2.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
449     label2.setText("Symbols");
450     label2.setOpaque(false);
```

```
451     label2.setFont(bold12);
452     label2.setTextBold();
453     cdl = new GDropList(winColors, 10, 70, 100, 140, 6, 15);
454     cdl.setItems(alpha, 0);
455     cdl.setLocalColorScheme(5);
456     cdl.addEventHandler(this, "alphaChooser");
457     GButton btnColor = new GButton(winColors, 120, 70, 70, 20, "Change");
458     btnColor.addEventHandler(this, "handleColorChooser");
459     GView view = new GView(winColors, 10, 100, 220, 20, JAVA2D);
460     label3color = new GLabel(winColors, 10, 100, 220, 20);
461     label3color.setTextAlign(GAlign.CENTER, GAlign.MIDDLE);
462     selectedColor = 0;
463     label3color.setText(alpha[selectedColor]);
464     label3color.setOpaque(false);
465     label3color.setFont(bold12);
466     label3color.setTextBold();
467     pg1 = view.getGraphics();
468     pg1.beginDraw();
469     pg1.background(alphaColors[selectedColor]);
470     pg1.endDraw();
471
472     GLabel label3stroke = new GLabel(winColors, 10, 130, 220, 20);
473     label3stroke.setTextAlign(GAlign.CENTER, GAlign.MIDDLE);
474     label3stroke.setText("Borders");
475     label3stroke.setOpaque(false);
476     label3stroke.setFont(bold12);
477     label3stroke.setTextBold();
478     GButton btnColorStroke = new GButton(winColors, 160, 160, 70, 20,
        ↪  "Change");
479     btnColorStroke.addEventHandler(this, "handleColorChooserStroke");
480     GView viewStroke = new GView(winColors, 10, 160, 140, 20, JAVA2D);
481     pg12 = viewStroke.getGraphics();
482     pg12.beginDraw();
483     pg12.background(ca.getColorStroke());
484     pg12.endDraw();
485
486     GButton randomize = new GButton(winColors, 50, 190, 140, 20);
487     randomize.setText("Randomize colors");
488     randomize.addEventHandler(this, "btnColorsRandomizeCallback");
489
490     GButton summit = new GButton(winColors, 75, 220, 90, 20);
491     summit.setText("OK");
492     summit.addEventHandler(this, "btnColorsOkCallback");
493   } else {
```

```
494     G4P.showMessage(this, "There is not a cellular automata, create new
        ↪  or open some one in File", "Cellular automata missed",
        ↪  G4P.ERROR_MESSAGE);
495   }
496 }
497
498 void createSimulatorCAWindow() {
499   if (isTS) {
500     if (txf2.getText() != "") {
501       nh = float(txfh.getText());
502       nw = float(txfw.getText());
503       println(nh);
504       println(nw);
505       winSimulatorCA = GWindow.getWindow(this, "SimulatorCA", 280, 0,
          ↪  displayWidth/2 - 280, displayHeight - 60, JAVA2D);
506       winSimulatorCA.setActionOnClose(G4P.CLOSE_WINDOW);
507       winSimulatorCA.performCloseAction();
508       h = nh*sdr1.getValueI();
509       w = nw*sdr1.getValueI();
510       ca.setScale(sdr1.getValueI());
511       ca.initialize(txf2.getText(), int(nw), int(nh));
512       ca.setOnlyTS(cbx1.isSelected());
513       posX = 0;
514       posY = 0;
515       dzoomX = 0;
516       dzoomY = 0;
517       zoom = 0;
518       onetime = true;
519       thread("generateNext");
520       println("Init Simulator");
521       winSimulatorCA.addDrawHandler(this, "win_simulatorca_draw");
522       winSimulatorCA.addMouseHandler(this, "SimulatorCAMouse");
523       winSimulatorCA.addOnCloseHandler(this, "closeSimulatorCACallback");
524     } else {
525       G4P.showMessage(this, "Enter a initial string or generate a random
          ↪  initial string", "Initial string missed", G4P.ERROR_MESSAGE);
526     }
527   } else {
528     G4P.showMessage(this, "There is not a cellular automata, create new
        ↪  or open some one in File", "Cellular automata missed",
        ↪  G4P.ERROR_MESSAGE);
529   }
530 }
531
532 void createStatisticsWindow() {
```

```
533   if (isTS) {
534     if (txf2.getText() != "") {
535       winStatistics = GWindow.getWindow(this, "Statistics", 280, 0, 800,
          ↪ displayHeight - 60, JAVA2D);
536       winStatistics.setActionOnClose(G4P.CLOSE_WINDOW);
537       cbx2 = new GCheckbox(winStatistics, 10, 0, 160, 20, "Show Legend");
538       cbx2.setSelected(false);
539       onetimePlot = true;
540       winStatistics.addDrawHandler(this, "win_statistics_draw");
541       winStatistics.addOnCloseHandler(this, "closeStatisticsCallback");
542     } else {
543       G4P.showMessage(this, "Enter a initial string or generate a random
          ↪ initial string", "Initial string missed", G4P.ERROR_MESSAGE);
544     }
545   } else {
546     G4P.showMessage(this, "There is not a cellular automata, create new
      ↪ or open some one in File", "Cellular automata missed",
      ↪ G4P.ERROR_MESSAGE);
547   }
548 }
549
550 public void sdr1CallBack(GSlider slider, GEvent event) {
551   if (ca !=  null && event.getType().trim().equals("RELEASED") &&
    ↪ sclValue != sdr1.getValueI()) {
552     ca.setScale(sdr1.getValueI());
553     h = nh*sdr1.getValueI();
554     w = nw*sdr1.getValueI();
555     sclValue = sdr1.getValueI();
556     pg = ca.render();
557   }
558 }
559
560 public void handleToggleControlEvents(GOption option, GEvent event) {
561   if (ca != null) {
562     if (grp1_a.isSelected())
563       ca.setShowStroke(true);
564     if (grp1_b.isSelected())
565       ca.setShowStroke(false);
566     pg = ca.render();
567   }
568 }
569
570 void saveJSON(GButton button, GEvent event) {
571   JSONObject newJSON = new JSONObject();
572   JSONObject tsJSON = new JSONObject();
```

```
573   JSONObject caJSON = new JSONObject();
574   tsJSON.setInt("P", gts.p);
575
576   JSONArray tsSigma = new JSONArray();
577   for (int i = 0; i < gts.getSymbols().length; i++)
578     tsSigma.setString(i, gts.getSymbols()[i]);
579   tsJSON.setJSONArray("Sigma", tsSigma);
580
581   JSONArray tsRules = new JSONArray();
582   for (int i = 0; i < gts.getRules().size(); i++)
583     tsRules.setString(i, gts.getRules().get(i));
584   tsJSON.setJSONArray("Rules", tsRules);
585
586   JSONArray caSigma = new JSONArray();
587   for (int i = 0; i < ca.getAlpha().length; i++)
588     caSigma.setString(i, ca.getAlpha()[i]);
589   caJSON.setJSONArray("Sigma", caSigma);
590
591   JSONArray caColors = new JSONArray();
592   for (int i = 0; i < alphaColors.length; i++)
593     caColors.setInt(i, alphaColors[i]);
594   caJSON.setJSONArray("Colors", caColors);
595
596   JSONArray caRules = new JSONArray();
597   for (int i = 0; i < gts.getCARules().size(); i++)
598     caRules.setString(i, gts.getCARules().get(i));
599   caJSON.setJSONArray("Rules", caRules);
600
601   newJSON.setJSONObject("ts", tsJSON);
602   newJSON.setJSONObject("ca", caJSON);
603   String pathName = G4P.selectOutput("Save as JSON...", ".json", "JSON
      ↪ Files");
604   if (pathName != null) {
605     if (pathName.trim().toLowerCase().endsWith(".json"))
606       saveJSONObject(newJSON, pathName);
607     else
608       saveJSONObject(newJSON, pathName + ".json");
609   }
610   filePrincipal.setCollapsed(true);
611 }
612
613 void openJSON(GButton button, GEvent event) {
614   JSONObject newJSON = new JSONObject();
615   JSONObject tsJSON = new JSONObject();
616   JSONObject caJSON = new JSONObject();
```

```
617
618   String pathName = G4P.selectInput("Open JSON...", ".json", "JSON
      ↪  Files");
619
620   if (pathName != null) {
621     if (ca != null) ca.reset();
622
623     if (pathName.trim().endsWith(".json")) {
624       newJSON = loadJSONObject(pathName);
625
626       if (!newJSON.isNull("ts") && !newJSON.isNull("ca")) {
627         tsJSON = newJSON.getJSONObject("ts");
628
629         if (winSimulatorCA != null) {
630           winSimulatorCA.close();
631           winSimulatorCA = null;
632         }
633
634         if (winNewTS != null) {
635           winNewTS.close();
636           winNewTS = null;
637         }
638
639         if (winColors != null) {
640           winColors.close();
641           winColors = null;
642         }
643
644         if (winStatistics != null) {
645           winStatistics.close();
646           winStatistics = null;
647         }
648
649         onetime = true;
650
651         int p = tsJSON.getInt("P");
652
653         JSONArray tsSigma = new JSONArray();
654         tsSigma = tsJSON.getJSONArray("Sigma");
655
656         JSONArray tsRules = new JSONArray();
657         tsRules = tsJSON.getJSONArray("Rules");
658
659         String[] rs = tsRules.getStringArray();
660         StringList appendants = new StringList();
```

```
661         StringList prodRulesList = new StringList();
662
663         for (int i = 0; i < rs.length; i++) {
664           prodRulesList.append(rs[i].trim());
665           String[] rn = rs[i].split("->");
666           if (rn.length > 1)
667             appendants.append(rn[1].trim());
668           else
669             appendants.append("");
670         }
671         pg = null;
672         ca = null;
673         gts = makeTagSystem(tsSigma.getStringArray(), p,
            ↪  appendants.array());
674         gts.setRules(prodRulesList);
675
676         caJSON = newJSON.getJSONObject("ca");
677
678         JSONArray caSigma = new JSONArray();
679         caSigma = caJSON.getJSONArray("Sigma");
680
681         JSONArray caRules = new JSONArray();
682         caRules = caJSON.getJSONArray("Rules");
683
684         JSONArray caColors = new JSONArray();
685         caColors = caJSON.getJSONArray("Colors");
686
687         ca = new CA(new StringList(caRules.getStringArray()), new
            ↪  StringList(caSigma.getStringArray()));
688         String tsMsg = "TS: R=" + appendants.array().length + " P=" + p;
689         String caMsg = "CA: R=" + gts.getCARules().size() + " Sigma=" +
            ↪  gts.getCAAlpha().size();
690         StringBuilder s = new StringBuilder();
691         s.append(tsMsg + "\n");
692         s.append(caMsg);
693         label1principal.setText(s.toString());
694         println(s.toString());
695         label1principal.setTextBold();
696         ca.setETS(gts.getSymbols());
697         alpha = ca.getAlpha();
698         alphaColors = caColors.getIntArray();
699         ca.setAlphaColors(alphaColors);
700         isTS = true;
701       } else {
```

```
702         G4P.showMessage(this, "This JSON file doesn't have required
              ↪  fields.", "No Data in JSON", G4P.ERROR_MESSAGE);
703       }
704     } else {
705       G4P.showMessage(this, "File type not supported.", "JSON missed",
            ↪  G4P.ERROR_MESSAGE);
706     }
707   }
708   filePrincipal.setCollapsed(true);
709 }
710
711 void generateRandomStringTS(GButton button, GEvent event) {
712   gts.setRandomLen(int(txf3.getText()));
713   txf2.setText(gts.randomInitialString());
714   winTSLengthRandom.close();
715   winTSLengthRandom = null;
716 }
717
718 void generateRandomStringCA(GButton button, GEvent event) {
719   nh = float(txfh.getText());
720   nw = float(txfw.getText());
721   ca.initialize(txf2.getText(), int(nw), int(nh));
722   txf2.setText(ca.randomInitialString());
723 }
724
725 void btnZoomCallback(GButton button, GEvent event) {
726   zoom = float(button.tag)/100;
727   zoomPanel.setCollapsed(true);
728 }
729
730 void createGUI() {
731   G4P.messagesEnabled(false);
732   G4P.setGlobalColorScheme(GCScheme.BLUE_SCHEME);
733   G4P.setMouseOverEnabled(false);
734   G4P.setDisplayFont("Montserrat", G4P.BOLD, 12);
735   surface.setTitle("GTS2CA 1.0");
736   surface.setLocation(0, 0);
737
738   GButton.useRoundCorners(false);
739
740   filePrincipal = new GPanel(this, 0, 0, 35, 20);
741   filePrincipal.setText("File     ...");
742   filePrincipal.setCollapsed(true);
743   filePrincipal.setDraggable(false);
744   GButton btnOpenTSwindow = new GButton(this, 0, 20, 60, 20);
```

```
745   btnOpenTSwindow.setText("Open");
746   btnOpenTSwindow.addEventHandler(this, "openJSON");
747   GButton btnNewTSwindow = new GButton(this, 0, 40, 60, 20);
748   btnNewTSwindow.setText("New");
749   btnNewTSwindow.addEventHandler(this, "btnNewTSCallback");
750   GButton btnSaveTSwindow = new GButton(this, 0, 60, 60, 20);
751   btnSaveTSwindow.setText("Save");
752   btnSaveTSwindow.addEventHandler(this, "saveJSON");
753   filePrincipal.addControl(btnNewTSwindow);
754   filePrincipal.addControl(btnSaveTSwindow);
755   filePrincipal.addControl(btnOpenTSwindow);
756
757   saveImg = new GPanel(this, 36, 0, 50, 20);
758   saveImg.setText("Export     ...");
759   saveImg.setCollapsed(true);
760   saveImg.setDraggable(false);
761   btnSaveImgPNG = new GButton(this, 0, 20, 100, 20);
762   btnSaveImgPNG.setText("PNG Image");
763   btnSaveImgPNG.addEventHandler(this, "btnSaveImgCallback");
764   btnSaveImgJPG = new GButton(this, 0, 40, 100, 20);
765   btnSaveImgJPG.setText("JPG Image");
766   btnSaveImgJPG.addEventHandler(this, "btnSaveImgCallback");
767   btnSaveImgTIFF = new GButton(this, 0, 60, 100, 20);
768   btnSaveImgTIFF.setText("TIFF Image");
769   btnSaveImgTIFF.addEventHandler(this, "btnSaveImgCallback");
770   btnSaveImgTXT = new GButton(this, 0, 80, 100, 20);
771   btnSaveImgTXT.setText("Text");
772   btnSaveImgTXT.addEventHandler(this, "btnSaveTextCallback");
773   saveImg.addControl(btnSaveImgPNG);
774   saveImg.addControl(btnSaveImgJPG);
775   saveImg.addControl(btnSaveImgTIFF);
776   saveImg.addControl(btnSaveImgTXT);
777
778   zoomPanel = new GPanel(this, 87, 0, 45, 20);
779   zoomPanel.setText("Zoom     ...");
780   zoomPanel.setCollapsed(true);
781   zoomPanel.setDraggable(false);
782
783   GButton[] btnZoom = new GButton[7];
784   for (int i = 0; i < 7; i++) {
785     btnZoom[i] = new GButton(this, 0, 20 + i*20, 45, 20);
786     btnZoom[i].setText(str(50 + i*25) + "%");
787     btnZoom[i].addEventHandler(this, "btnZoomCallback");
788     btnZoom[i].tag = str(-50 + i*25);
789     zoomPanel.addControl(btnZoom[i]);
```

```
790     }
791
792     label1principal = new GLabel(this, 20, 40, 240, 60);
793     label1principal.setTextAlign(GAlign.CENTER, GAlign.MIDDLE);
794     label1principal.setText("There is not a Tag System, press File to enter
        ↪  a Tag System");
795     label1principal.setFont(plain16);
796
797     GLabel label21 = new GLabel(this, 20, 130, 100, 20);
798     label21.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
799     label21.setText("Simulation");
800     label21.setOpaque(false);
801     label21.setTextBold();
802
803     txf2 = new GTextField(this, 20, 160, 240, 20);
804     txf2.setPromptText("Initial String");
805     txf2.setText("");
806
807     GButton btnRandomTS = new GButton(this, 20, 190, 115, 20);
808     btnRandomTS.setText("Random TS Str");
809     btnRandomTS.addEventHandler(this, "btnLengthRandomCallback");
810
811     GButton btnRandomCA = new GButton(this, 145, 190, 115, 20);
812     btnRandomCA.setText("Random CA Str");
813     btnRandomCA.addEventHandler(this, "generateRandomStringCA");
814
815     int l2sx = 20, l2sy = 220;
816     GLabel label2 = new GLabel(this, l2sx, l2sy, 70, 20);
817     label2.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
818     label2.setText("Scale");
819     label2.setOpaque(false);
820
821     sdr1 = new GSlider(this, l2sx, l2sy, 140, 80, 10);
822     sdr1.setLimits(2, 1, 10);
823     sdr1.setLocalColorScheme(6);
824     sdr1.setOpaque(false);
825     sdr1.setNbrTicks(10);
826     sdr1.setShowLimits(true);
827     sdr1.setShowValue(true);
828     sdr1.setShowTicks(true);
829     sdr1.setStickToTicks(true);
830     sdr1.setEasing(10);
831     sdr1.setRotation(0.0, GControlMode.CENTER);
832     sdr1.addEventHandler(this, "sdr1CallBack");
833
```

```
834   int l3tx = 180, l3ty = 220;
835   GLabel label3 = new GLabel(this, l3tx, l3ty, 70, 20);
836   label3.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
837   label3.setText("Borders");
838   label3.setOpaque(false);
839
840   togG1Options = new GToggleGroup();
841   grp1_a = new GOption(this, l3tx, l3ty + 20, 120, 20);
842   grp1_a.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
843   grp1_a.setText("Yes");
844   grp1_a.addEventHandler(this, "handleToggleControlEvents");
845   grp1_b = new GOption(this, l3tx, l3ty + 40, 120, 20);
846   grp1_b.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
847   grp1_b.setText("No");
848   grp1_b.addEventHandler(this, "handleToggleControlEvents");
849   togG1Options.addControl(grp1_a);
850   togG1Options.addControl(grp1_b);
851   grp1_b.setSelected(true);
852
853   GLabel label4 = new GLabel(this, 20, 300, 50, 20);
854   label4.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
855   label4.setText("Cells:");
856   label4.setOpaque(false);
857   txfw = new GTextField(this, 70, 300, 110, 20);
858   txfw.setPromptText("No. of Cells");
859   txfw.setText("300");
860
861   GLabel label5 = new GLabel(this, 20, 330, 90, 20);
862   label5.setTextAlign(GAlign.LEFT, GAlign.MIDDLE);
863   label5.setText("Evolutions:");
864   label5.setOpaque(false);
865   txfh = new GTextField(this, 105, 330, 140, 20);
866   txfh.setPromptText("No. of Evolutions");
867   txfh.setText("1000");
868
869   cbx1 = new GCheckbox(this, 20, 360, 180, 18, "Only TS Alphabet
      ↪  Strings");
870   cbx1.setSelected(false);
871
872   GButton btnColors = new GButton(this, 70, 390, 140, 20);
873   btnColors.setText("Colors Setup");
874   btnColors.addEventHandler(this, "btnColorsCallback");
875
876   GButton btnStatistics = new GButton(this, 80, 420, 120, 20);
877   btnStatistics.setText("See Stats");
```

```
878   btnStatistics.addEventHandler(this, "btnStatisticsCallback");
879
880   GButton btnSimulatorCA = new GButton(this, 80, 450, 120, 20);
881   btnSimulatorCA.setText("Run Simulation");
882   btnSimulatorCA.addEventHandler(this, "btnSimulatorCACallback");
883 }
```