Life Attractors - Cellular automaton

Gael Hernández Solís

December 17, 2023

Contents

1 Introducción

En el vasto universo del pensamiento computacional y la teoría de la complejidad, el Juego de la Vida emerge como un fenómeno extraordinario que trasciende las barreras de lo abstracto para manifestarse en un tapiz digital de patrones sorprendentes y dinámicas intrincadas. Este juego, concebido por el matemático británico John Horton Conway a principios de la década de 1970, es una ventana hacia la autopoiesis de la vida digital, donde células binarias obedecen reglas simples para crear patrones complejos e inesperados.

El Juego de la Vida pertenece a la categoría de los "automatones celulares", una clase de modelos matemáticos que representan sistemas dinámicos discretos. La semilla de esta fascinante creación surgió en 1970 cuando Conway, un genio matemático, buscaba un modelo de autómata celular con reglas simples pero capaz de generar complejidad. Su visión culminó en un tablero infinito dividido en células cuadradas, cada una de las cuales podía estar viva (1) o muerta (0).

El corazón palpitante del Juego de la Vida reside en las reglas B/S, que determinan el destino de cada célula en función de su entorno. La notación B/S se refiere a la supervivencia y la natalidad de las células. Por ejemplo, B3/S23 indica que una célula muerta nacerá si tiene exactamente tres vecinos vivos, y una célula viva sobrevivirá si tiene dos o tres vecinos vivos. Esta simplicidad aparente engendra una complejidad emergente que ha desconcertado a matemáticos y entusiastas de la computación durante décadas.

Si bien el Juego de la Vida puede parecer inicialmente un pasatiempo matemático, sus aplicaciones se extienden mucho más allá de las fronteras de la diversión intelectual. Los patrones generados por este juego han demostrado ser útiles en el diseño de algoritmos, criptografía y simulaciones científicas. Además, la noción de complejidad emergente que encarna el Juego de la Vida tiene implicaciones en campos tan diversos como la biología, la inteligencia artificial y la teoría de sistemas complejos.

Desde su creación, el Juego de la Vida ha sido el epicentro de la investigación y la experimentación en el ámbito de los autómatas celulares. La comunidad científica ha descubierto patrones estables, osciladores, y naves espaciales que desafían las expectativas y amplían nuestra comprensión de la complejidad computacional. La colaboración global ha llevado al descubrimiento de estructuras de datos notables y ha generado un vasto conjunto de conocimientos que sigue evolucionando.

Más allá de su utilidad práctica, el Juego de la Vida despierta la imaginación y la reflexión sobre la naturaleza de la vida y la complejidad. Los patrones intrincados que emergen en el tablero digital invitan a contemplar la belleza de las estructuras autónomas y a reflexionar sobre las similitudes entre la vida biológica y la artificial. ¿Es posible que las reglas simples que gobiernan este juego virtual compartan alguna conexión fundamental con los principios que guían la evolución en el mundo real?

Las reglas B/S son la columna vertebral del Juego de la Vida, y su comprensión precisa es esencial para prever la evolución de las células en el tablero. Las células nacen si tienen un número específico de vecinos vivos, y sobreviven si su entorno satisface ciertas condiciones. Modificar estas reglas puede conducir a comportamientos radicalmente diferentes, desde patrones estáticos hasta caóticas explosiones de actividad.

La implementación eficiente del Juego de la Vida implica la aplicación de algoritmos cuidadosamente diseñados. El algoritmo estándar, conocido como "vecinos contados", evalúa el estado de cada célula basándose en la cuenta de sus vecinos vivos. Sin embargo, las optimizaciones avanzadas, como el uso de estructuras de datos eficientes y técnicas de paralelización, han permitido simular tableros masivos con mayor velocidad y menor consumo de recursos computacionales.

Aunque el Juego de la Vida es un ejemplo célebre de autómatas celulares unidimensionales, la extensión a autómatas celulares bidimensionales agrega capas adicionales de complejidad y diversidad. Estos modelos, a menudo representados por matrices 2D, han sido objeto de estudio para comprender la variabilidad en la dinámica emergente y explorar nuevas formas de representar sistemas complejos.

Aunque el Juego de la Vida sigue reglas deterministas, su comportamiento a veces puede parecer caótico. Esta aparente contradicción se resuelve al considerar la "sensibilidad a las condiciones iniciales", una característica común en sistemas caóticos. Pequeñas variaciones en la disposición inicial de células pueden resultar en evoluciones drásticamente distintas, lo que añade una dimensión adicional de complejidad al juego.

El Juego de la Vida presenta un vasto espacio de configuraciones, desde patrones estáticos y oscilantes hasta naves espaciales y estructuras caóticas. Conceptos como "atractores" describen conjuntos de condiciones iniciales que llevan a configuraciones estables, mientras que la "estabilidad" se refiere a la capacidad de ciertos patrones para persistir indefinidamente en el tiempo, aportando un matiz adicional a la riqueza de resultados posibles.

Aunque Conway dejó un legado duradero con el Juego de la Vida, la comunidad científica ha continuado explorando variantes y extensiones. Desde reglas modificadas hasta dimensiones adicionales, estas variaciones han generado nuevos desafíos y perspectivas, ampliando el alcance del juego original y consolidándolo como un campo activo de investigación.

2 Desarrollo

A continuación, se mostrará primero el simulador donde se implementó el Juego de la Vida bajo la regla R(Smin,Smax,Nmin,Nmax), con este fue posible calcular todos los nodos y conexiones correspondientes de los diagramas de ciclos (atractores) de los autómatas celulares B3/S23 ó R(2,3,3,3) y B7/S2 ó R(2,2,7,7). Mostraré la forma en la que se calculó y se graficó con ayuda de MATLAB los diagramas para configuraciones de tamaño 2x2, 3x3, 4x4 y 5x5. Después de esto, mostraré los patrones encontrados en los atractores y en las reglas para dar conclusiones finales.

Cabe mencionar que este simulador se realizó con Processing 4.3, el cuál es un entorno de programación y desarrollo creativo que se utiliza para crear aplicaciones y obras de arte interactivas, principalmente en el ámbito de la programación visual y generativa, el lenguaje utilizado fue Java 8.

2.1 Vista general de simulador

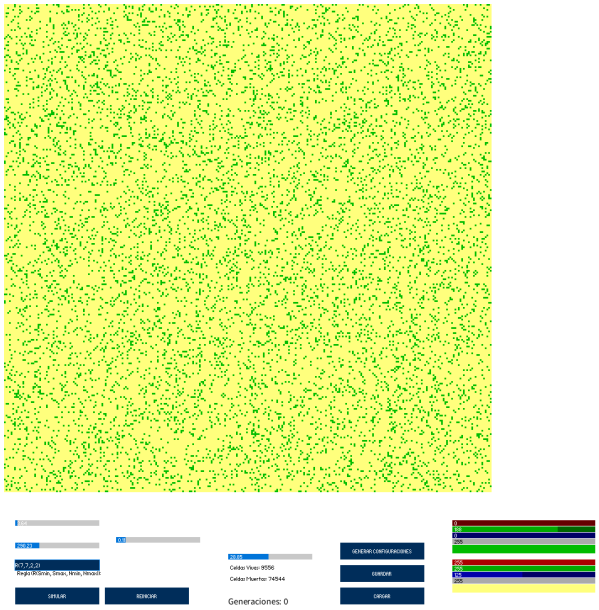


Figure 1: Vista general del simulador

Como podemos ver, en el simulador tenemos los siguientes botones y secciones:
Botones

- Simular: Sirve para empezar con la simulación indefinida de acuerdo a la configuración que se tenga
- Reiniciar: Sirve para borrar la generación actual del simulador y dejar todas las celdas muertas.
- Guardar: Sirve para salvar en un archivo .txt el estado de la generación que se tenga en el simulador
- Cargar: Sirve para levantar de un archivo .txt el estado de celdas vivas y muertas que se tengan y debe ser igual al tamaño de nuestra cuadrícula del grid.
- Generar configuraciones: Sirve para calcular todas las combinaciones que se pueden tener en el tamaño de cuadrícula que se tenga actual, obtiene el valor decimal de dicha configuración, calcula la siguiente generación y obtiene su valor en decimal, ambos valores los guarda en un archivo .txt llamado configuraciones.

Secciones

- Colores: Nos permite indicar el color que queramos de la celda muerta y viva en nuestra simulación.
- Tamaño de cuadro: Permite indicar el tamaño en pixeles que queremos que tengan nuestras celdas, desde un pixel hasta un tamaño de 100 pixeles
- Tamaño de la cuadrícula: Permite indicar el tamaño de nuestra cuadrícula, desde 10x10 cuadros hasta un tamaño de pantalla completa, es decir, 1920x1080 cuadros
- Velocidad: Permite ajustar la velocidad con la que queremos que vayan avanzando las generaciones, desde 0.1 hasta 60 fotogramas por segundo
- Regla: Ingresar la regla que queramos en el formato R Smin,Smax,Nmin,Nmax
- Llenado: Slider que rellena la cuadrícula actual con el número de celdas vivas que queramos de manera aleatoria, desde un 0 porciento hasta un 100 porciento de celdas vivas.

Indicadores

- Celdas muertas: Indica el número de celdas muertas que se encuentren en la pantalla y se actualiza automático.
- Celdas vivas: Indica el número de celdas vivas que se encuentren en la pantalla y se actualiza automático.
- Generaciones: Indica la generación actual de nuestro autómata y se actualiza solo.

2.2 Regla R(2,3,3,3) ó B3/S23

Esta regla establece que una célula muerta cobrará vida si tiene exactamente tres vecinos vivos, y una célula viva seguirá viva si tiene dos o tres vecinos vivos, pero morirá en cualquier otro caso

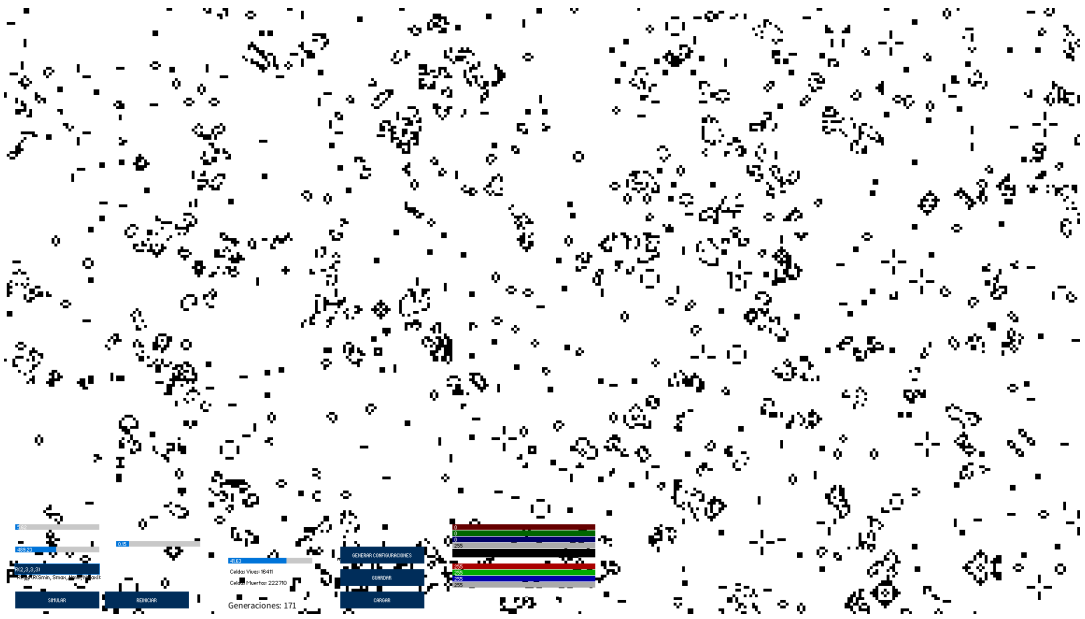


Figure 2: Regla B3/S23 con 490x490 celdas tras 171 generaciones con 15 porciento de celdas vivas iniciales

Patrones encontrados

Después de haber simulado dicho autómata en la figura anterior, se encontraron los siguientes patrones, los clasificaremos por su comportamiento y se describirán brevemente:

- **Osciladores**

Figure 3: Conjunto de osciladores

En esta figura podemos observar 4 osciladores separados perfectamente para que no se interrumpa su comportamiento, el oscilador en sí son tres celdas vivas contiguas la cual tiene un periodo de dos, es decir, sólo tiene dos estados en cada paso.

(a) Primer estado del oscilador

(b) Segundo estado del oscilador

Figure 4: Oscilador

Este es el otro oscilador que encontré el cuál también es de periodo dos y en las figuras se muestran cuáles son.

- **Gliders**

  Bien sabemos que un glider es una estructura que se desplaza por el tablero, para este claso sólo encontré un glider y es conocido como "planeador".

Figure 5: Planeador hacia arriba

Este planeador está configurado de tal forma que se iba desplazando en diagonal hacia arriba a la derecha.

Figure 6: Planeador hacia abajo

Este planeador está configurado de tal forma que se iba desplazando en diagonal hacia abajo a la derecha.

- **Estructuras muertas**

  Dentro de las estructuras estáticas, es decir, que no cambian a pesar de que alguna estructura los altere, encontramos las siguientes:

Figure 7: Bloque

Figure 8: Loaf

Figure 9: Beehive

Figure 10: Boat

Figure 11: Tub

Figure 12: Long boat

Se realizó la investigación de cada uno y se encontró que esos son sus nombres originales, y estos son los que pude identificar dentro de la simulación, obviamente deben existir más.

- **Pulsar**

Figure 13: Pulsar

Este es un pulsar que encontramos, es decir, un oscilador periódico que muestra un patrón repetitivo a lo largo del tiempo, volviendo a su estado original después de un número específico de generaciones, es una figura que se repetía mucho en varios espacios, sin embargo, como abarca mucho espacio, era inevitable que chocara con otras estructuras y se desintegrara.

- **Estructuras espontáneas**

  Con estructuras espontáneas me refiero a la formación de patrones caóticos o a la aparición de estructuras inesperadas que chocaban con todo a su paso para generar nuevas estructuras, a continuación enseñe alñgunas que encontré, sin embargo no es como que sean de alguna forma en particular.

Figure 14: Patrón caótico

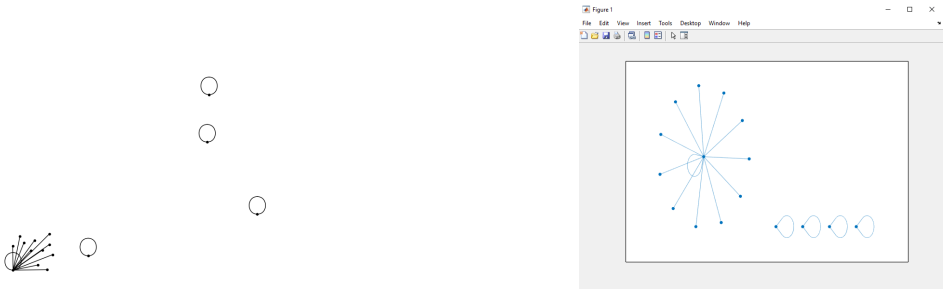
Figure 15: Patrón caótico

Una vez mostrados estos patrones de acuerdo a su comportamiento, podemos ver que muchos son equivalentes, es decir, que a pesar de verse diferentes, hacen el mismo comportamiento, lo podemos ver con los dos gliders que se mostraron, los cuales van en diferente dirección, sin embargo, si nos damos cuenta es la misma figura solo que con ligeras variaciones en sus celdas vivas y muertas, nosotros podríamos hacer otros dos gliders que en vez de moverse hacia la derecha, se muevan hacia la izquierda tanto para arriba y para abajo.
Así como encontramos estos patrones, existen muchísimas más estructuras que podríamos observar con esta misma regla, sin embargo, depende mucho de las condiciones iniciales, de cuántas generaciones dejemos que avancen, o para el mejor caso, nosotros probar con estructuras iniciales y ver qué da como resultado.

Ahora mostraremos los atractores generados para los tamaños solicitados y hablaremos de ellos un poco.

Atractores

- Tamaño 2x2



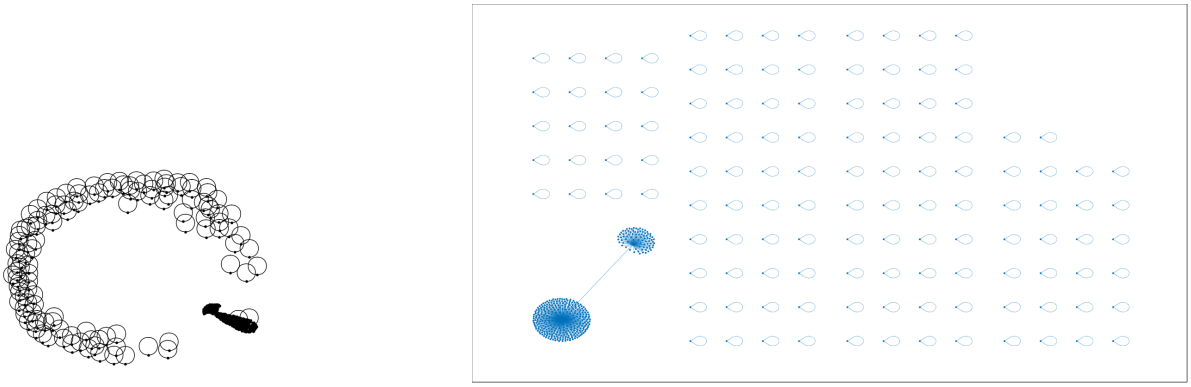
(a) Atractor hecho con Python

(b) Atractor hecho con MATLAB

Figure 16: Atractores de tamaño 2x2

Para este caso, el atractor tiene 16 nodos en total, casi todas las configuraciones iniciales dieron en su siguiente generación un estado completo de células muertas excepto los casos de confguraciones iniciales con valor decimal de 3, 5, 10 y 12. Esto quiere decir entonces que la mayoría de los nodos convergen en uno mismo, y su resultado es cero, excepto estos cuatro casos que son nodos separados con una única evolución, como se puede observar en el diagrama. Hablando de la topología del atractor, el graficador de python no muestra una topología específica, es sencillo, por otro lado, MATLAB nos lo mostró con una topología de tipo estrella, esperemos que se siga observando esta topología en gráficas próximas.

- Tamaño 3x3



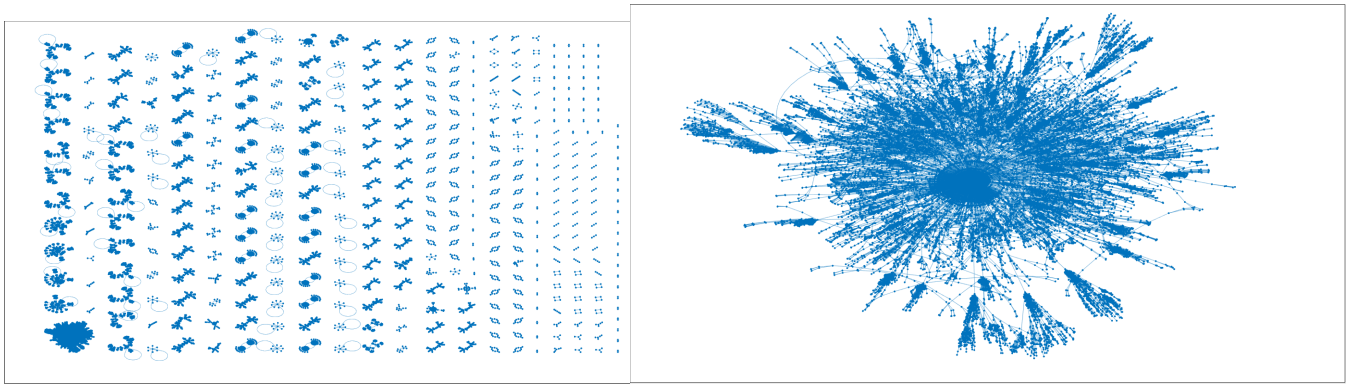
(a) Atractor hecho con Python

(b) Atractor hecho con MATLAB

Figure 17: Atractores de tamaño 3x3

Para este caso, ya estamos hablando de tener 512 nodos, y todos los nodos que se encuentran en forma de anillo en la gráfica de python son aquellos nodos únicos que no convergen en uno mismo, por otro lado, en la parte baja de la gráfica está una masa de nodos negra, esos nodos son los que convergen en 0. Del lado de la gráfica de MATLAB, de igual manera los nodos solitarios son nodos únicos y la figura de la pate inferior izquierda son los que convergen en 0 y en el número 511 en decimal claro. La topología de igual manera, tiene que ver más que nada con el algoritmo empleado para la graficación, por el lado de Python utilicé el algortimo de Fruchterman-Reingold que trata de que los nodos estén distribuidos de manera uniforme y las aristas tengan longitudes aproximadamente iguales.

- Tamaño 4x4

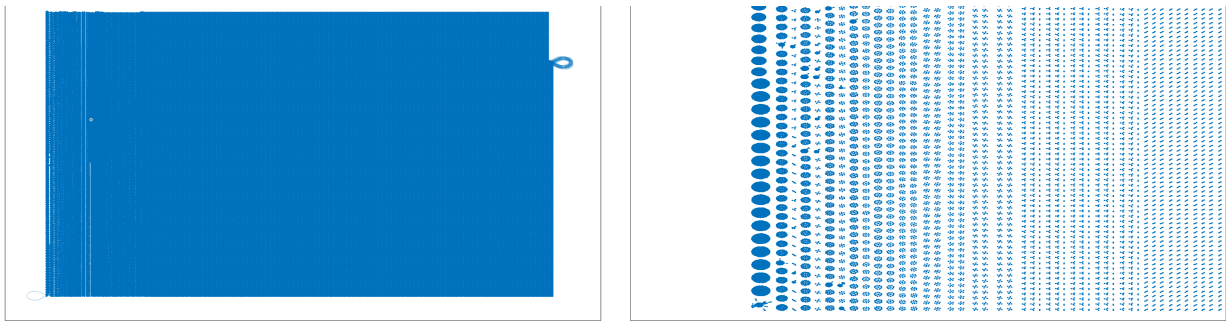


(a) Atractor hecho con MATLAB (b) Acercamiento a la estructura más caótica

Figure 18: Atractores de tamaño 4x4

Para este atractor ya estamos hablando de tener 65536 nodos, por parte de la graficación en python, se utilizaron como 8 diferentes algoritmos para poder visualizar el grafo de una manera diferente, sin embargo, en todos lo muestra sin distinción alguna, como un círculo completamente negro. Se usó también la aplicación Gephi para graficarlos todos pero también mostró un cuadrado completamente negro. Por otro lado, con MATLAB podemos visualizar una gran diversidad de topologías, es propio de MATLAB pero aún así nos damos cuenta que ya no todas las configuraciones iniciales convergen a un sólo punto que era 0 en los anteriores casos, entonces esto nos dice que ya a partir de un tamaño de cuadrícula de 4x4 que no es para nada gran diferencia comparándolo con el de 3x3 o 2x2, ya obtenemos una gran diversidad de evoluciones, y si visualizamos bien las estructuras más complejas de la parte inferior izquierda de la pantalla, sonará muy burdo pero se parece demasiado a los atractores vistos en clase, claro es normal eso pero me sorprende en lo particular.

- Tamaño 5x5



(a) Atractor hecho con MATLAB (b) Acercamiento

Figure 19: Atractores de tamaño 5x5

Finalmente, para el último atractor, hablamos de que tiene 33554432 millones de nodos, el graficador de python con ningún algoritmo lo pudo realizar en tiempo adecuado, llevó dos horas y ni así lo terminaba de graficar, por otro lado, con MATLAB también tardó demasiado en graficarlo, por lo que tuve que seleccionar una muestra aleatoria de 500000 nodos de los 33 millones para graficarlo, tardó una hora pero sí lo graficó. Lamentablemente no podemos ni pensar en cómo se vería el grafo completo, la diversidad de topologías que muchas son equivalentes, sin embargo, no quiere decir que son lo mismo. Ya son cálculos que a pesar del poder computacional que existe, aún tarda mucho tiempo en realizarlos y mucho más en graficarlos, pero con este último ejemplo nos damos cuenta de la gigantezca complejidad que puede surgir gracias a esta regla y en general del Juego de la Vida en espacios demasiado pequeños, estamos hablando de 25 celdas, es impresionante.

Finalmente para terminar con el estudio de esta regla, graficaremos la densidad de unos y la entropía de Shannon a través de grandes generaciones para concluir en qué momento la regla se estabiliza y bajo

qué condiciones iniciales.

Gráficas de Densidad y Entropía

- **10 porciento de celdas vivas iniciales**

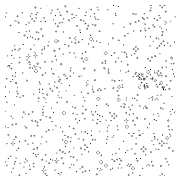
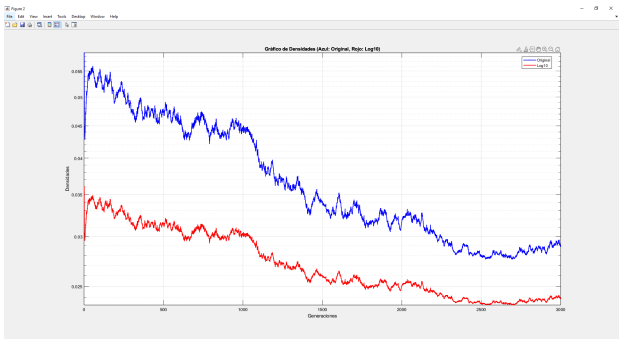
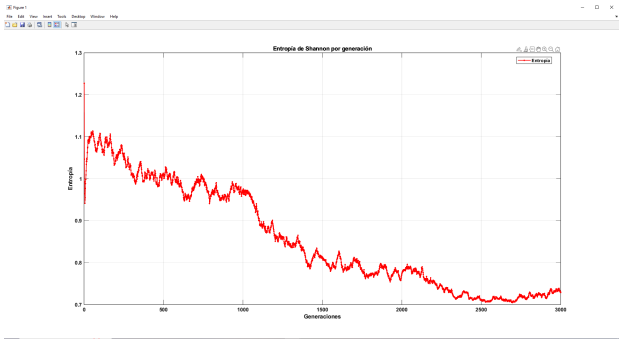


Figure 20: Regla R(2,3,3,3) con 355x355 celdas tras 3000 generaciones



(a) Densidad de celdas vivas

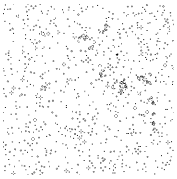


(b) Entropía de Shannon

Figure 21: Gráficas

En estas gráficas para un 10 porciento de celdas vivas iniciales, para la parte de la densidad de unos, parece ser que se estabiliza a partir de la generación 2500 pero es impreciso porque como la cantidad de celdas vivas es menor y las estructuras están algo separadas, en caso de que surja una estructura caótica y empiece a chocar con otras, empezará a incrementar y decrementar drásticamente la cantidad de unos, por lo que para este porcentaje se ve a partir de esa generación pero es impreciso. Lo mismo para la entropía, se podría decir que se empezará a estabilizar a partir de la 2500, esto quiere decir que la distribución de estados en el sistema alcanzará un estado más predecible o menos caótico.

- **25 porciento de celdas vivas iniciales**



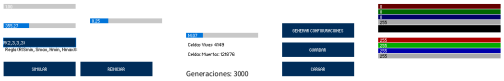
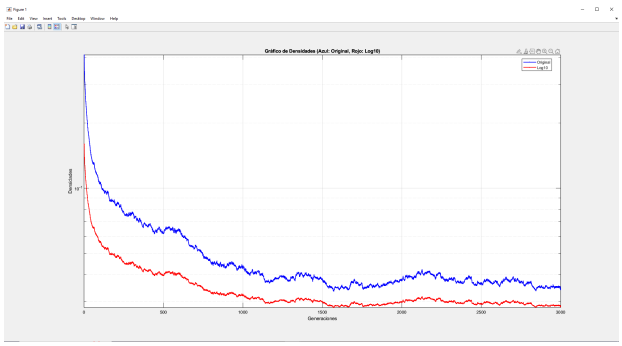
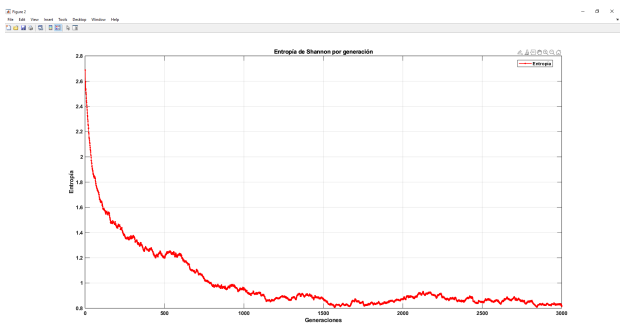


Figure 22: Regla R(2,3,3,3) con 355x355 celdas tras 3000 generaciones



(a) Densidad de celdas vivas

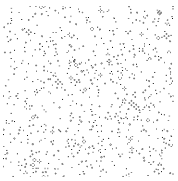


(b) Entropía de Shannon

Figure 23: Gráficas

Para este caso inicial del 25 porciento, en la gráfica de densidad de unos ya se ve un mejor comportamiento, podemos ver que se empieza a estabilizar a partir de la generación 1000, ya de ahí tiene ligeras perturbaciones pero esto es debido a las estructuras caóticas que llegan a surgir y alteran el sistema. En el lado de la entropía, parece ser que de igual manera se empezaría a estabilizar a partir de la generación 2500, sin embargo, que no esté estabilizada la gráfica de esta propiedad quiere decir que el comportamiento del sistema es caótico, eso en un caso drástico, pero en el nuestro, denota un comportamiento muy poco predecible.

- **50 porciento de celdas vivas iniciales**



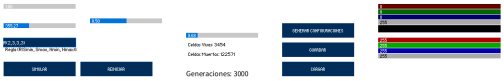
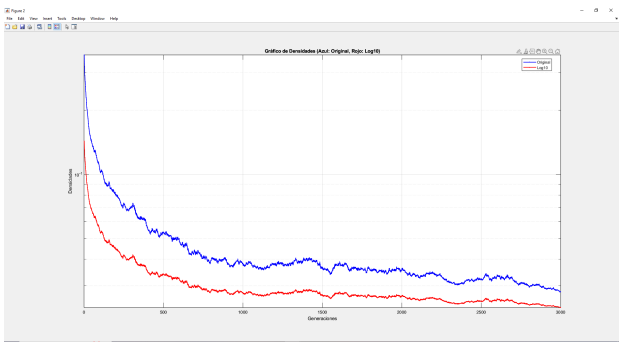


Figure 24: Regla R(2,3,3,3) con 355x355 celdas tras 3000 generaciones



(a) Densidad de celdas vivas

(b) Entropía de Shannon

Figure 25: Gráficas

Finalmente, para este caso inicial de celdas vivas, podemos observar que en la gráfica de densidad se empieza a estabilizar a partir de la generación 1000, cuando llega a la 2500, empieza a decrementar la cantidad de celdas vivas, esto podría significar que el sistema estaría llegando a una mejor estabilización pues también hay que considerar la gráfica de su entropía, que se mantiene muy constante a partir de la generación 1000 también, esto nos dice que el sistema ya no es tan caótico, sería cuestión de ver más generaciones para dar un análisis más contundente.

Pero a partir de estos tres casos de análisis, puedo decir que el sistema bajo esta regla del Juego de la Vida funciona de dos maneras, cuando tenemos un porcentaje de celdas vivas del 1 porciento al 5 porciento, y un porcentaje del 80 al 90 porciento, el sistema se estabiliza de inmediato pues las celdas que sobrevivan será muy pocas y no nacerán más. El otro caso se encuentra en porcentajes del 6 al 79 porciento, donde hablar de que el sistema se vuelva predecible o menos caótico es muy poco probable, más que nada por las estructuras espontáneas que ya mencionamos, estas alteran todo el sistema y no hay nada que las detenga, por eso depende de estas más que nada.

2.3 Regla R(7,7,2,2) ó B2/S7

Las celdas en la cuadrícula bajo esta regla evolucionarán de la siguiente manera, una célula muerta con exactamente dos vecinos vivos cobrará vida y una célula viva con siete vecinos vivos sobrevivirá.

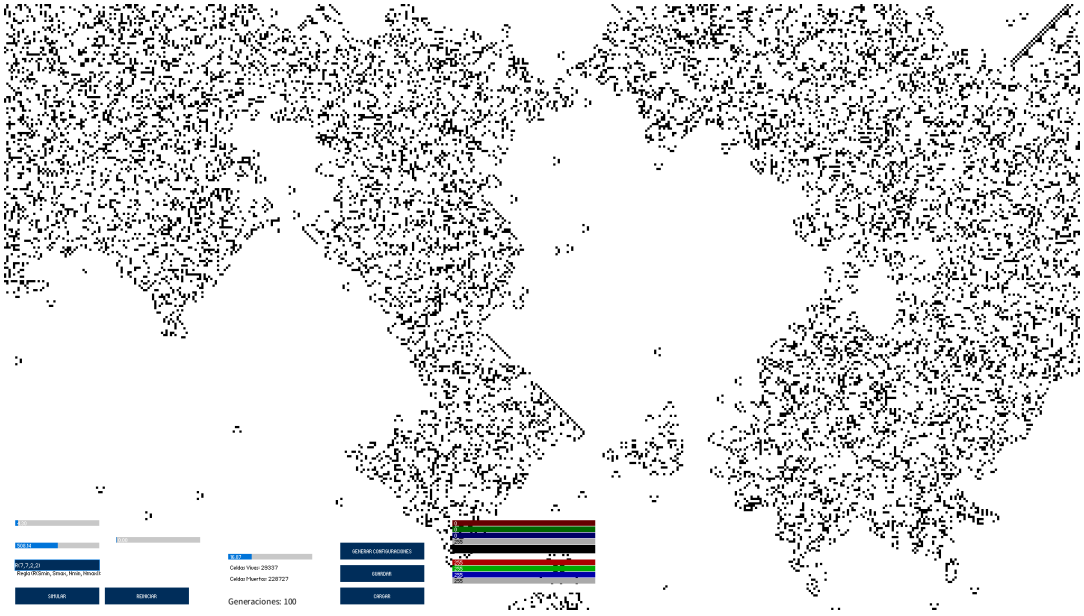


Figure 26: Regla B2/S7 con 510x10 celdas tras 100 generaciones con 1 porciento de celdas vivas iniciales

Patrones encontrados

- Osciladores

(a) Primer estado del oscilador

(b) Segundo estado del oscilador

Figure 27: Oscilador

(a) Primer estado del oscilador

(b) Segundo estado del oscilador

Figure 28: Oscilador

Estos dos osciladores son los únicos que encontré que fueran estables, es decir, permanecen por sí solos oscilando hasta que alguna estructura externa interrumpa su periodo. Encontré otras estructuras que tenían varios estados al paso de las generaciones, sin embargo, al paso de tres o cuatro generaciones desaparecían, una de ellas son tres celdas vivas contiguas.

- Estructuras muertas

  No encontré estructuras estáticas dentro de esta regla, el autómata en todo momento se encontraba cambiando, lo que podría decir es que estructuras muertas que ya vimos en la anterior regla, como lo es la colmena (beehive) o tubo (tub), específicamente estas dan lugar a osciladores no estables o estructuras dinámicas que ya veremos.

- Estructuras dinámicas

(a) Barrera que avanza en diagonal

(b) Nave espacial

Figure 29: Estructuras dinámicas

La primera figura es una barrera que se formó a partir de patrones complejos, logré hacer una barrera a partir de la estructura muerta llamada "bloque", que de igual manera, son estructuras que avanzan diagonalmente, y la rotar de tal forma que avance hacia cualquier dirección, esta lo que hace es limpiar las estructuras que se vaya encontrando, para el caso de la barrera formada a partir del bloque, se rompe fácilmente al chocar con otras estructuras. La segunda figura es una nave espacial que avanza de igual manera hacia cualquier dirección de manera recta, la podemos configurar para que avance hacia la derecha o hacia abajo, por lo que tiene otras tres estructuras equivalentes.

(a) Puffer train

(b) Puffer train dejando rastro

Figure 30: Estructura dinámica

Para esta estructura que investigando encontré que se llama "puffer train", es una estructura que genera un rastro de estructuras más pequeñas a medida que se mueve como se ve en la segunda imagen, de igual manera, la podemos configurar de tal forma que avance hacia abajo, hacia la derecha o izquierda, tiene más figuras equivalentes.

(a) Iniciador de una barrera

(b) Barreras

Figure 31: Estructuras dinámicas

No encontré el nombre de esta estrcutura, tiene forma de una paloma, lo que hace es un generador de barreras, tanto delanteras como traseras, y avanza en diagonal, estas barreras sí limpian estructuras simples pues al chocar con una estructura, rompe la primera barrera pero de inmediato la estructura compleja está generando otra y así constantemente hasta abarcar todo el espacio.

(a) Tipo de puffer train

(b) Después de un tiempo

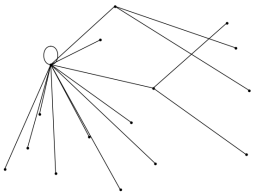
Figure 32: Estructuras complejas

Estas estructuras son parecidas al puffer train pues de igual manera van dejando estructuras simples a su paso, y estas a su vez evolucionan a otros patrones más complejos, por lo que estas últimas estructuras que hemos mencionado podríamos decir que son patrones complejos también pues van formando demasiadas estructuras a la vez, lo que hace que abarquen todo el espacio.

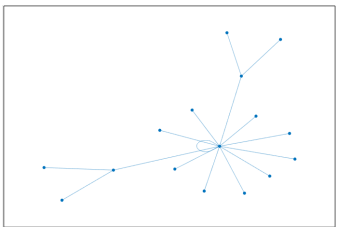
De esta forma, hay muchas más estructuras simples que con ligeras variaciones en sus celdas vivas o muertas, pueden generar estos patrones que ya explicamos y vimos, además de que con rotar la estructura y volverla a generar, podemos lograr que su dirección sea otra, ya sea que avance diagonalmente o en linea recta, pero son estructuras muy simples que generan patrones muy complejos, esto es propio de esta regla.

Atractores

- Tamaño 2x2



(a) Atractor hecho con Python



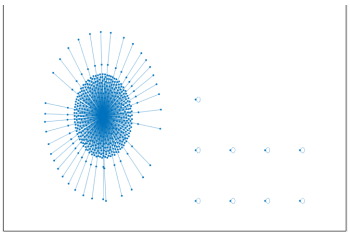
(b) Atractor hecho con MATLAB

Figure 33: Atractores de tamaño 2x2

De nueva cuenta, tenemos un atractor con 16 nodos, casi todas las configuraciones iniciales dieron en su siguiente generación un estado completo de células muertas excepto los casos de confguraciones iniciales con valor decimal de 1, 2, 4 y 8. Hablando de la topología del atractor, el graficador de python muestra una topología de árbol, es sencillo, por otro lado, con MATLAB se ve una topología de tipo estrella, la mayoría de los nodos que convergen en uno, son los que convergen en 0, los otros dos casos convergen en un valor decimal de 6 y 9.

- Tamaño 3x3

(a) Atractor hecho con Python

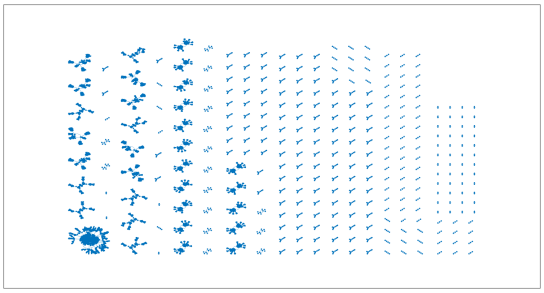


(b) Atractor hecho con MATLAB

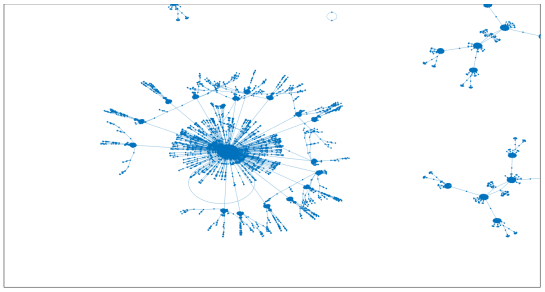
Figure 34: Atractores de tamaño 3x3

Este atractor tiene 512 nodos y la mayoría convergen a 0, los nodos que tiene unión consigo mismos convergen a estados únicos, considerando la cantidad de configuraciones iniciales que son posibles, se me hace muy poco diversa esta regla para este tamaño, no hay mucho que decir, sería cuestión de ver cómo será el atractor para tamaños más grandes. La topología de los atractores tampoco cambia mucho, es más que nada por el algoritmo que utilza cada aplicación.

- Tamaño 4x4



(a) Atractor hecho con MATLAB

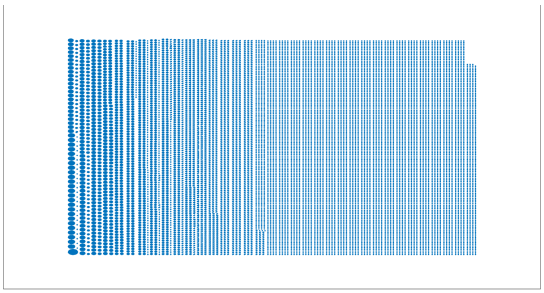


(b) Acercamiento a la estructura más caótica

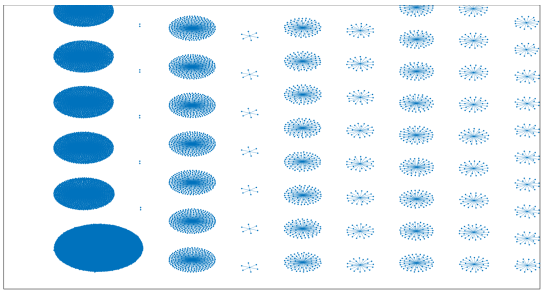
Figure 35: Atractores de tamaño 4x4

65536 nodos, la libería usada para graficar el atractor en python llamada "networkx" junto con los algoritmos que trae para la visualizació del mismo, no pudieron hacer que todos los nodos se vieran bien, por lo que a partir de aquí usaremos MATLAB. En este atractor ya podemos decir que tenemos una gran diversidad de topologías, ya no todas las configuraciones iniciales convergen a un solo estado como en los dos casos anteriores. Comparándolo con el atractor de la anterior regla, a simple vista y con ayuda de nuestro archivo .txt para la conexión de los nodos, la regla B3/S23 tiene una mayor diversidad para este tamaño de cuadrícula, hay mucha más variedad de topologías, esto nos dice que esa regla B3/S23 es más diversa y rica en términos de la variedad de patrones que puede generar, sus condiciones de supervivencia y nacimiento son más permisivas, lo que lleva a una mayor complejidad y una gama más amplia de comportamientos emergentes.

- Tamaño 5x5



(a) Atractor hecho con MATLAB



(b) Acercamiento

Figure 36: Atractores de tamaño 5x5

33554432 millones de nodos, de igual manera se seleccionó de una muestra aleatoria de 500000 nodos de los 33 millones para graficarlo. No se puede apreciar en la figura todas las estructuras del atractor pues son demasiadas, sin embargo, también podemos concluir que muchas topologías son equivalentes, sin embargo, no quiere decir que converjan hacia los mismos estados. También acercándonos y viendo las estructuras del atractor, podemos decir que sí, la regla anterior tiene mucha más diversidad en patrones y evoluciones pero esta también es una regla interesante.

Para terminar con este trabajo, graficaremos la densidad de unos y la entropía de Shannon como lo hicimos con la anterior la regla para saber cómo se comporta esta regla a través de las generaciones.

Gráficas de Densidad y Entropía

- **2 porciento de celdas vivas iniciales**

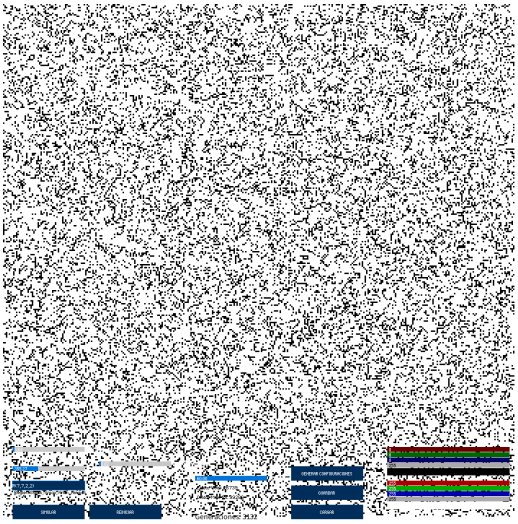
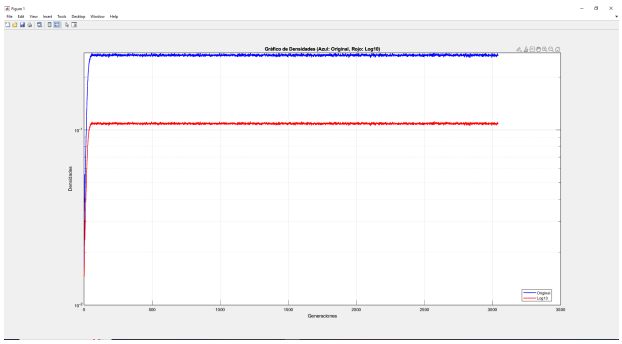
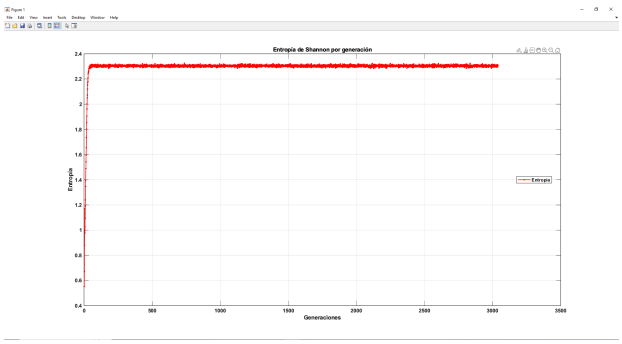


Figure 37: Regla R(7,7,2,2) con 355x355 celdas tras 3000 generaciones



(a) Densidad de celdas vivas



(b) Entropía de Shannon

Figure 38: Gráficas

Para este caso inicial del 2 porciento, en la gráfica de densidad de celdas vivas podemos ver que parece estabilizarse de inmediato con el paso de las generaciones, como desde la generación 50, al igual que la gráfica de su entropía, sin embargo, por la parte visual que yo tuve al paso de las generaciones, no había patrones o estructuras que yo pudiera identificar, todo el sistema era un caos, decir que su entropía se estabiliza tras cierta generación quiere decir que el autómata se vuelve predecible o menos caótico y no es el caso de esta regla, podría decir que, este autómata siempre va a llegar a ser caótico sin immportar el porcentaje inicial, claro si tenemos un 100 porciento inicial de unos no sucederá nada, pero para otros casos, siempre llegará a ser así, eso podríamos decir que es predecible pues ya sabemos a qué llegará siempre.

- **10 porciento de celdas vivas**

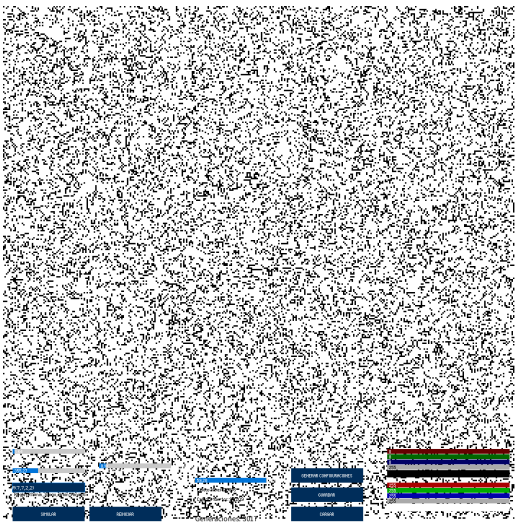
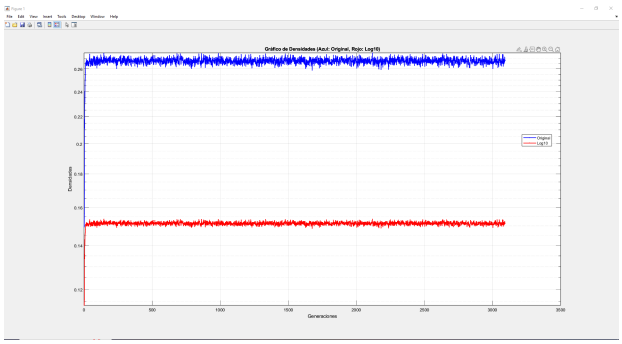
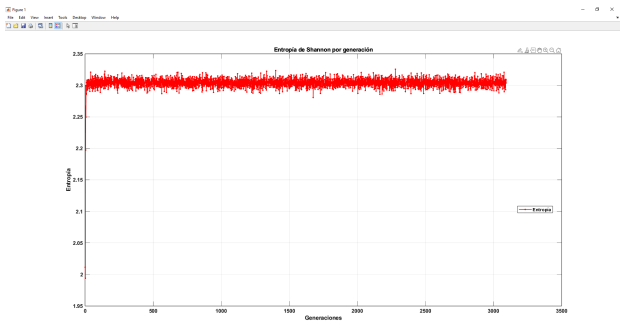


Figure 39: Regla R(7,7,2,2) con 355x355 celdas tras 3000 generaciones



(a) Densidad de celdas vivas



(b) Entropía de Shannon

Figure 40: Gráficas

Para el caso inicial del 10 porciento, tanto en la gráfica de densidad como en la de entropía, de igual manera parece ser que se estabilizan al corto paso de las generaciones, la única diferencia claro es que los valores de densidades aumentaron y la entropía ligeramente aumentó también pero se debe más que nada a la condición inicial. Nuevamente podemos decir que el autómata se vuelve caótico más que nada por los patrones y estructuras que ya mencionamos anteriormente, y como todos están en constante chico gracias a la naturaleza de la regla, provoca estos comportamientos en las gráficas.

- **50 porciento de celdas vivas**

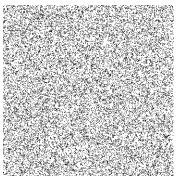
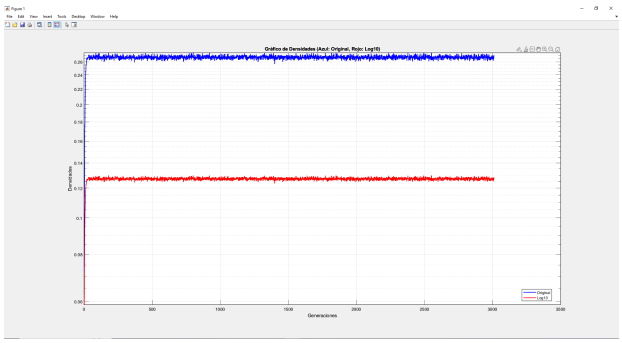


Figure 41: Regla R(7,7,2,2) con 355x355 celdas tras 3000 generaciones



(a) Densidad de celdas vivas

(b) Entropía de Shannon

Figure 42: Gráficas

Finalmente para este caso inicial, el valor de las densidades y las entropías se mantuvieron iguales comparados con el de 10 porciento, esto nos dice que de igual manera, para condiciones de celdas vivas iniciales mayores al 50 porciento y menores a un 99 porciento, el comportamiento al que se llegará es el mismo, un comportamiento caótico sin distinción de patrones, eso ya es propio de la regla, los patrones interesantes surgen cuando tratamos el espacio de célular de manera manual, generando figuras bien distanciadas para ver cómo se comportan con el tiempo, si tenemos demasiadas celdas vivas, no tendrá mucha aplicación para algún estudio esta regla.

3 Código Fuente de Processing 4.3

```
import controlP5.*;
import java.io.*;

int cols, rows;
float[][] grid;
boolean simulationRunning = false;
int sMin, sMax, nMin, nMax;
String ruleInput = "R(7,7,2,2)";
ControlP5 cp5;

int cellsPerSideSliderValue = 100;
int cellSizeSliderValue = 5;
float probabilityDropdownValue = 0.9;
int generationCount = 0;
float simulationSpeed = 1.0;
int aliveCellColor = color(0);
int deadCellColor = color(255);
PrintWriter output;
PrintWriter entropyOutput;

void setup() {
        fullScreen();
```

```
        cp5 = new ControlP5(this);
        output = createWriter("densidad.txt");
        entropyOutput = createWriter("entropia.txt");
        initializeGrid(cellsPerSideSliderValue, cellsPerSideSliderValue);

        // Interfaz
        createInputFields();
        createSimulateButton();
        createReiniciarButton();
        createGridSizeSlider();
        createCellSizeSlider();
        createProbabilityDropdown();
        createColorPickers();
        createStatusLabel();
        createSpeedSlider();
        createGuardarButton();
        createCargarButton();
        createConfigButton();
}

void draw() {
        background(255);
        drawGrid();
        updateStatusLabels();

        if (simulationRunning && frameCount % int(60 / simulationSpeed) == 0) {
                updateGrid();
                generationCount++;

                float density = calculateDensity();
                output.println(generationCount + "\t" + density);
                float entropy = calculateShannonEntropy();
                entropyOutput.println(generationCount + "\t" + entropy);
        }

        fill(0);
        textSize(16);
        text("Generaciones:-" + generationCount, 400, height - 10);
}

float calculateShannonEntropy() {

        int[] configCounts = new int[9];


        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {
                        int neighbors = countNeighbors(i, j);
                        configCounts[neighbors]++;
                }
        }


        float entropy = 0;
        float totalCells = cols * rows;

        for (int count : configCounts) {
                if (count > 0) {
                        float probability = count / totalCells;
                        entropy -= probability * log(probability) / log(2);
                }
        }

        return entropy;
}


float calculateDensity() {
        int countOnes = countAliveCells();
        int countZeros = cols * rows - countOnes;

        if (countZeros == 0) {
                return Float.POSITIVE_INFINITY;  // Evitar la divisi n por cero
        }

        return (float) countOnes / countZeros;
}

void exit() {
        // Cerrar el archivo al salir del programa
        output.flush();
        output.close();
        entropyOutput.flush();
        entropyOutput.close();
        super.exit();
}

void createColorPickers() {
        cp5.addColorPicker("aliveCellColor")
        .setPosition(800, height - 160)
        .setColorValue(aliveCellColor)
        .setLabel("Color-de-celdas-vivas")
        .getCaptionLabel().setVisible(false);

        cp5.addColorPicker("deadCellColor")
        .setPosition(800, height - 90)
        .setColorValue(deadCellColor)
        .setLabel("Color-de-celdas-muertas")
        .getCaptionLabel().setVisible(false);
}

void controlEvent(ControlEvent theEvent) {
        if (theEvent.isFrom("aliveCellColor")) {
                aliveCellColor = cp5.get(ColorPicker.class, "aliveCellColor").getColorValue();
        } else if (theEvent.isFrom("deadCellColor")) {
                deadCellColor = cp5.get(ColorPicker.class, "deadCellColor").getColorValue();
        }
}

void createGuardarButton() {
        cp5.addButton("guardarButton")
        .setPosition(600, height - 80)
        .setSize(150, 30)
        .setCaptionLabel("Guardar")
        .onClick(e -> guardarConfiguracion("guardada.txt"));
}
```

```
void createConfigButton(){
        cp5.addButton("generarConfiguracionesButton")
        .setPosition(600, height - 120)
        .setSize(150, 30)
        .setCaptionLabel("Generar Configuraciones")
        .onClick(e -> generarYGuardarConfiguraciones());

}

void createCargarButton() {
        cp5.addButton("cargarButton")
        .setPosition(600, height - 40)
        .setSize(150, 30)
        .setCaptionLabel("Cargar")
        .onClick(e -> cargarConfiguracion("cargada.txt"));
}

void guardarConfiguracion(String nombreArchivo) {
        String[] contenido = new String[rows];
        for (int j = 0; j < rows; j++) {
                String fila = "";
                for (int i = 0; i < cols; i++) {
                        fila += str(int(grid[i][j]));
                }
                contenido[j] = fila;
        }
        saveStrings(nombreArchivo, contenido);
}

void cargarConfiguracion(String nombreArchivo) {
        String[] lineas = loadStrings(nombreArchivo);
        if (lineas.length == rows) {
                for (int j = 0; j < rows; j++) {
                        String fila = lineas[j];
                        for (int i = 0; i < cols; i++) {
                                grid[i][j] = float(fila.charAt(i) - '0');
                        }
                }
        } else {
                println("Error: La configuraci n en el archivo no coincide con el tama o de la
                     cuadr cula.");
        }
}


void createSpeedSlider() {
        cp5.addSlider("speedSlider")
        .setPosition(400, height - 100)
        .setSize(150, 10)
        .setRange(0.1, 60.0)
        .setValue(simulationSpeed)
        .setColorBackground(color(200))
        .setLabel("Generaciones por segundo")
        .setLabelVisible(true)
        .onChange(e -> simulationSpeed = cp5.get(Slider.class, "speedSlider").getValue());
}

void createStatusLabel() {
        cp5.addTextlabel("vivasLabel")
        .setPosition(400, height - 80)
        .setText("Celdas Vivas: 0")
        .setColorValue(color(0));

        cp5.addTextlabel("muertasLabel")
        .setPosition(400, height - 60)
        .setText("Celdas Muertas: 0")
        .setColorValue(color(0));
}

void updateStatusLabels() {
        int vivas = countAliveCells();
        int muertas = cols * rows - vivas;

        cp5.get(Textlabel.class, "vivasLabel").setText("Celdas Vivas: " + vivas);
        cp5.get(Textlabel.class, "muertasLabel").setText("Celdas Muertas: " + muertas);
}


int countAliveCells() {
        int count = 0;
        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {
                        count += grid[i][j];
                }
        }
        return count;
}

void mousePressed() {
        if (!simulationRunning) {
                int col = floor(mouseX / cellSizeSliderValue);
                int row = floor(mouseY / cellSizeSliderValue);

                if (col >= 0 && col < cols && row >= 0 && row < rows) {
                        grid[col][row] = 1 - grid[col][row];
                }
        }
}


void keyPressed() {
        if (key == '-') {
                simulationRunning = !simulationRunning;
        }
}

void createInputFields() {
        cp5.addTextfield("ruleInput")
        .setPosition(20, height -90)
        .setSize(150, 20)
        .setText("R(7,7,2,2)")
        .onChange(e -> parseRule());

        cp5.addTextlabel("ruleLabel")
        .setPosition(20, height - 70)
```

```
        .setText("Regla-(R(Smin,-Smax,-Nmin,-Nmax)):")
        .setColorValue(color(0));
}

void createSimulateButton() {
        cp5.addButton("simulateButton")
        .setPosition(20, height - 40)
        .setSize(150, 30)
        .setCaptionLabel("Simular")
        .onClick(e -> {
                parseRule();
                simulationRunning = true;
                generationCount = 0;
                initializeGrid(cellsPerSideSliderValue, cellsPerSideSliderValue);
        });
}

void createReiniciarButton() {
        cp5.addButton("reiniciarButton")
        .setPosition(180, height - 40)
        .setSize(150, 30)
        .setCaptionLabel("Reiniciar")
        .onClick(e -> {
                generationCount = 0;
                initializeGrid(cellsPerSideSliderValue, cellsPerSideSliderValue);
        });
}

void createGridSizeSlider() {
        cp5.addSlider("cellsPerSideSlider")
        .setPosition(20, height - 120)
        .setSize(150, 10)
        .setRange(3, 1000)
        .setValue(cellsPerSideSliderValue)
        .setColorBackground(color(200))
        .onChange(e -> resizeGrid());
}

void createCellSizeSlider() {
        cp5.addSlider("cellSizeSlider")
        .setPosition(20, height - 160)
        .setSize(150, 10)
        .setRange(1, 100)
        .setValue(cellSizeSliderValue)
        .setColorBackground(color(200))
        .onChange(e -> resizeGrid());
}

void createProbabilityDropdown() {
        cp5.addSlider("probabilitySlider")
        .setPosition(200, height - 130)
        .setSize(150, 10)
        .setRange(0, 1)
        .setValue(probabilityDropdownValue)
        .setColorBackground(color(200))
        .setLabel("Proporci n-de-unos")
        .setLabelVisible(true)
        .onChange(e -> {
                probabilityDropdownValue = cp5.get(Slider.class, "probabilitySlider").getValue();
                fillGridWithProbability(probabilityDropdownValue);
        });
}

void initializeGrid(int initialCols, int initialRows) {
        cols = initialCols;
        rows = initialRows;
        grid = new float[cols][rows];
        fillGridWithProbability(probabilityDropdownValue);
        generationCount = 0;
}

void drawGrid() {
        noStroke();

        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {
                        float x = i * cellSizeSliderValue;
                        float y = j * cellSizeSliderValue;

                        if (grid[i][j] == 1) {
                                fill(aliveCellColor);
                        } else {
                                fill(deadCellColor);
                        }

                        rect(x, y, cellSizeSliderValue, cellSizeSliderValue);
                }
        }
}

void updateGrid() {
        float[][] next = new float[cols][rows];

        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {
                        float state = grid[i][j];
                        int neighbors = countNeighbors(i, j);

                        if (state == 0 && neighbors >= nMin && neighbors <= nMax) {
                                next[i][j] = 1;
                        } else if (state == 1 && neighbors >= sMin && neighbors <= sMax) {
                                next[i][j] = 1;
                        } else {
                                next[i][j] = 0;
                        }
                }
        }

        grid = next;
}

int countNeighbors(int x, int y) {
        int count = 0;
        for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                        int col = (x + i + cols) % cols;
```

```
                        int row = (y + j + rows) % rows;
                        count += grid[col][row];
                }
        }
        count -= grid[x][y];
        return count;
}


void parseRule() {
        String[] parts = splitTokens(cp5.get(Textfield.class, "ruleInput").getText(), "(),");
        if (parts.length >= 5) {
                sMin = constrain(parseInt(parts[1]), 0, 8);
                sMax = constrain(parseInt(parts[2]), 0, 8);
                nMin = constrain(parseInt(parts[3]), 0, 8);
                nMax = constrain(parseInt(parts[4]), 0, 8);
        }
}

void resizeGrid() {
        cellsPerSideSliderValue = int(cp5.get(Slider.class, "cellsPerSideSlider").getValue());
        cellSizeSliderValue = int(cp5.get(Slider.class, "cellSizeSlider").getValue());
        initializeGrid(cellsPerSideSliderValue, cellsPerSideSliderValue);
}

void fillGridWithProbability(float probability) {
        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {
                        float rand = random(1);
                        grid[i][j] = (rand < probability) ? 1 : 0;
                }
        }
}

void generarYGuardarConfiguraciones() {
        // Abrir el archivo para escritura
        PrintWriter outputConfiguraciones = createWriter("configuraciones.txt");

        // Iterar sobre todas las configuraciones posibles
        float totalConfiguraciones = pow(2, cols * rows);
        for (int configDecimal = 0; configDecimal < totalConfiguraciones; configDecimal++) {
                // Convertir el n mero decimal a binario y llenar la cuadr cula
                String configBinaria = Integer.toBinaryString(configDecimal);
                llenarCuadriculaDesdeConfiguracionBinaria(configBinaria);

                // Calcular la primera generaci n con la regla actual
                parseRule(); // Asegurarse de tener la regla actualizada
                simulationRunning = true;
                generationCount = 0;
                updateGrid();

                // Obtener los valores decimales de la configuraci n inicial y la primera generaci n
                int valorInicial = configDecimal;
                int valorPrimeraGeneracion = convertirConfiguracionAValorDecimal();

                // Escribir los valores en el archivo
                outputConfiguraciones.print(valorInicial + "\t" + valorPrimeraGeneracion);

                // Hacer un salto de l nea despu s de cada configuraci n
                outputConfiguraciones.println();
        }

        // Cerrar el archivo
        outputConfiguraciones.flush();
        outputConfiguraciones.close();
}

void llenarCuadriculaDesdeConfiguracionBinaria(String configBinaria) {
        // Asegurarse de tener una cuadr cula v lida
        initializeGrid(cols, rows);

        // Rellenar la cuadr cula con la configuraci n binaria
        int index = configBinaria.length() - 1;
        for (int i = cols - 1; i >= 0; i--) {
                for (int j = rows - 1; j >= 0; j--) {
                        if (index >= 0) {
                                grid[i][j] = configBinaria.charAt(index) - '0';
                                index--;
                        }
                }
        }
}

int convertirConfiguracionAValorDecimal() {
        int valorDecimal = 0;
        int base = 1;
        for (int i = cols - 1; i >= 0; i--) {
                for (int j = rows - 1; j >= 0; j--) {
                        valorDecimal += grid[i][j] * base;
                        base *= 2;
                }
        }
        return valorDecimal;
}
```

4 Código Fuente de MATLAB (Densidad y Log10)

```
data = load('C:\Users\Gael-Hernandez-Solis\Desktop\JV\densidad.txt');
min_value = min(data(:, 2));
min_value = abs(min_value) + 1;

figure;

semilogy(data(:, 1), data(:, 2), 'b.-', 'LineWidth', 1.5);
hold on;

semilogy(data(:, 1), log10(data(:, 2) + min_value), 'r.-', 'LineWidth', 1.5);
```

```
grid on;

xlabel('Generaciones');
ylabel('Densidades');
title('Grafico de Densidades (Azul: Original, Rojo: Log10)');

legend('Original', 'Log10', 'Location', 'Best');
```

5 Código Fuente de MATLAB (Entropía de Shannon)

```
data = load('C:\Users\Gael-Hernandez-Solis\Desktop\JV\entropia.txt');

generaciones = data(:, 1);
entropiaS = data(:, 2);

figure;

plot(generaciones, entropiaS, 'r.-', 'LineWidth', 1.5, 'MarkerSize', 10);

grid on;

xlabel('Generaciones');
ylabel('Entropia');
title('Entropia de Shannon por generacion');


set(gca, 'FontName', 'Arial', 'FontSize', 12, 'FontWeight', 'bold');
set(gcf, 'Color', 'w');
legend('Entrop\'{i}a', 'Location', 'Best');
```

6 Código Fuente de MATLAB (Atractores)

```
data = dlmread('C:\Users\Gael-Hern ndez-Sol s\Desktop\JV\77225x5.txt');

tamano_muestra = min(400000, size(data, 1));

indices_muestra = randperm(size(data, 1), tamano_muestra);
data_muestra = data(indices_muestra, :);

nodos_enteros = unique(data_muestra(:));
mapeo_nodos = containers.Map(nodos_enteros, 1:length(nodos_enteros));
data_muestra(:, 1) = cell2mat(values(mapeo_nodos, num2cell(data_muestra(:, 1))));
data_muestra(:, 2) = cell2mat(values(mapeo_nodos, num2cell(data_muestra(:, 2))));

G = graph(data_muestra(:, 1), data_muestra(:, 2));


h = plot(G, 'Layout', 'force', 'NodeLabel', {});

saveas(gcf, 'grafo_visualizacion.png');
```

7 Conclusiones

El estudio detallado de las reglas B3/S23 y B2/S7 en el simulador del Juego de la Vida ha proporcionado una visión profunda sobre el comportamiento dinámico y las propiedades emergentes de estos sistemas. La Regla B3/S23, hablando de comportamiento global y estabilidad, vimos que es muy permisiva en términos de supervivencia y nacimiento además de tener una rica complejidad. La variedad de patrones, incluyendo osciladores, estructuras estáticas y dinámicas, resalta su capacidad para generar comportamientos emergentes diversos. Hablando de la Regla B2/S7, esta tiene condiciones más restrictivas, ha mostrado una dinámica más compleja pero con una menor variedad de patrones concisos. Las configuraciones iniciales tienden a desaparecer o evolucionar hacia estados caóticos con mayor rapidez. Esta regla no favorece la formación de osciladores estables, aunque ciertos patrones oscilatorios pueden emerger inicialmente, tienden a desaparecer con el tiempo.

La entropía de Shannon para la primera regla, que mide la variabilidad en las distribuciones de celdas, sugiere un comportamiento dinámico y cambiante. La diversidad de patrones y la mayor densidad de unos para ciertas condiciones iniciales indican una evolución continua y compleja del sistema. Bajo la regla B2/S7, la entropía de Shannon puede estabilizarse más rápidamente junto con su densidad de unos, inidicando que sin importar las condiciones iniciales que le demos al autómata, siempre llegará al caos pero se mantendrá así siempre al paso de las generaciones, no surgirá un cambio en el sistema.

La regla B3/S23 es valiosa para explorar la complejidad y la diversidad en los autómatas celulares. Su capacidad para generar una amplia gama de patrones la hace adecuada para investigaciones en teoría de la complejidad y ciencias computacionales. Esta regla es propicia para la creación de "criaturas" dinámicas, incluyendo osciladores y naves espaciales, lo que la hace fascinante para la investigación en la simulación de sistemas autónomos.

La regla B2/S7 podría ser valiosa en contextos donde la estabilidad y la simplicidad son prioritarias, como en modelado de sistemas estacionarios o estáticos, esto claro, como ya se explicó, es en los casos cuando tratamos la configuración inicial de manera manual, y no de manera aleatoria.

Explorar otras reglas y comparar sus comportamientos podría enriquecer aún más el entendimiento del Juego de la Vida y sus aplicaciones potenciales en modelado y simulación.

Este proyecto no solo contribuye al entendimiento de las reglas específicas en el Juego de la Vida, sino que también proporciona una base sólida para futuras investigaciones y aplicaciones en áreas como la simulación de sistemas complejos y la modelización de fenómenos dinámicos.

8 Referencias

References

[1] The Global Dynamics of Cellular Automata; An Atlas of Basin of Attraction Fields of One-Dimensional Cellular Automata. Addison-Wesley, 1992.

[2] Sfairopoulos, K. (2023, 14 septiembre). Cellular automata in D dimensions and ground states of spin models in $(D + 1)$ dimensions. *arXiv.org*. `https://arxiv.org/abs/2309.08059`

[3] Search — ARXIV e-print Repository. (s. f.). `https://arxiv.org/search/?query=celular+rules&searchtype=all&source=header`

[4] Simple networks on complex cellular automata: From de Bruijn diagrams to jump-graphs. In: Swarm Dynamics as a Complex Network, Springer, (I. Zelinka and G. Chen Eds.), chapter 12, pages 241-264, 2018.