

Morales García Christian Arturo

Conway's Game of Life

Computing Selected Topics

Supervisor: Dr. Martinez Juarez Genaro

2021

Contents

Índice general	1
Contents	1
1 Introducción de Game of Life	3
2 Desarrollo de simulador de Game of Life	4
2.1 El juego	4
2.2 Implementación del juego de la vida en código	4
2.3 Tecnologías utilizadas para el desarrollo del juego de la vida	4
2.4 Primera iteración	4
2.5 Segunda iteración	5
2.6 Tercera iteración	6
3 Funcionamiento de Simulador Game of Life	7
3.1 Interfaz para configuración inicial	7
3.2 interfaz para cargar archivos	7
3.3 Interfaz con una configuración inicial	8
3.4 Pantalla principal de la simulación	8
3.5 Pantalla de simulación con células introducidas	9
3.6 Vista de simulación con células evolucionando	10
3.7 Vista de la gráfica con densidad de células	10
3.8 Interfaz de guardado de configuración actual de la simulación .	11
3.9 Vista de la configuración de las células	11
3.10 Pruebas con una matriz de 5000x5000	13
4 Modificación a la simulación	17
4.1 Interfaz de configuración inicial	17
4.2 Calculo de la Entropía	17
4.3 Código de implementación de Game of life	19
5 Conclusiones	31
6 Referencias	31
7 Introducción de Atractores	32
8 Desarrollo	32
8.1 El juego	32
8.2 Creación de atractores usando el juego de la vida	33
8.3 Tecnologías utilizadas para el desarrollo del juego de la vida	33
8.4 Primera iteración	33
8.5 Segunda iteración	36
8.6 Tercera iteración	38
9 Funcionamiento	39
9.1 Ventana principal de configuración de simulador de Game of life y generador de atractores	40
9.2 Ventana de visualización de grafo	41

9.3 Vista de archivos de texto generados por el generador de atractores	41
10 Pruebas de funcionamiento	42
10.1 Universo de 4x4 B3/S23	42
10.2 Universo de 2x2 B2/S3	45
10.3 Universo de 3x3 B2/S3	46
10.4 Universo de 4x4 B2/S3	47
10.5 Universo de 2x2 B2/S7	54
10.6 Universo de 3x3 B2/S7	55
10.7 Universo de 4x4 B2/S7	56
10.8 Código de implementación	64
11 Conclusiones finales	67
12 Referencias	68

1 Introducción de Game of Life



Figure 1: John Horton Conway

El Juego de la vida es un autómata celular diseñado por el matemático británico John Horton Conway en 1970.

John Horton Conway (Liverpool, 26 de diciembre de 1937 - Princeton, Nueva Jersey, 11 de abril de 2020)¹²³ fue un prolífico matemático británico, especialista en la teoría de grupos (teoría de grupos finitos), teoría de nudos, teoría de números, teoría de juegos y teoría de códigos. Se formó en Arratia BHI. Entre los matemáticos aficionados, quizás fue más conocido por su teoría de juegos combinatorios, en particular por ser el creador en 1970 del juego de la vida.

Hizo su primera aparición pública en el número de octubre de 1970 de la revista Scientific American, en la columna de juegos matemáticos de Martin Gardner. Desde un punto de vista teórico, es interesante porque es equivalente a una máquina universal de Turing, es decir, todo lo que se puede computar algorítmicamente se puede computar en el juego de la vida.

Desde su publicación, ha atraído mucho interés debido a la gran variabilidad de la evolución de los patrones. Se considera que el Juego de la vida es un buen ejemplo de emergencia y autoorganización. Es interesante para científicos, matemáticos, economistas y otros observar cómo patrones complejos pueden provenir de la implementación de reglas muy sencillas.

El Juego de la vida tiene una variedad de patrones reconocidos que provienen de determinadas posiciones iniciales. Poco después de la publicación, se descubrieron el pentaminó R, el planeador o caminador (en inglés, glider, conjunto de células que se desplazan) y el explosionador (células que parecen formar la onda expansiva de una explosión), lo que atrajo un mayor interés hacia el juego. Contribuyó a su popularidad el hecho de que se publicó justo cuando se estaba lanzando al mercado una nueva generación de miniordenadores baratos, lo que significaba que se podía jugar durante horas en máquinas que, por otro lado, no se utilizarían por la noche.

Para muchos aficionados, el juego de la vida solo era un desafío de programación y una manera divertida de usar ciclos de la CPU. Para otros, sin embargo,

el juego adquirió más connotaciones filosóficas.

2 Desarrollo de simulador de Game of Life

2.1 El juego

Se trata de un juego de cero jugadores, lo que quiere decir que su evolución está determinada por el estado inicial y no necesita ninguna entrada de datos posterior. El "tablero de juego" es una malla plana formada por cuadrados (las "células") que se extiende por el infinito en todas las direcciones. Por tanto, cada célula tiene 8 células "vecinas", que son las que están próximas a ella, incluidas las diagonales. Las células tienen dos estados: están "vivas" o "muertas" (o "encendidas" y "apagadas"). El estado de las células evoluciona a lo largo de unidades de tiempo discretas (se podría decir que por turnos). El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente.

Todas las células se actualizan simultáneamente en cada turno, siguiendo estas reglas:

1. Una célula muerta con exactamente 3 células vecinas vivas "nace" (es decir, al turno siguiente estará viva).
2. Una célula viva con 2 o 3 células vecinas vivas sigue viva, en otro caso muere (por "soledad" o "superpoblación").

2.2 Implementación del juego de la vida en código

2.3 Tecnologías utilizadas para el desarrollo del juego de la vida

- Python. Como lenguaje de programación para desarrollo con simplicidad, versatilidad y rapidez.
- Pygame. Al ser un conjunto de módulos del lenguaje Python permiten la creación de videojuegos en dos dimensiones de una manera sencilla.
- Matplot. Permitió el desarrollo de gráficas en tiempo real, al ser una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

2.4 Primera iteración

En esta parte se comenzó conociendo el entorno de trabajo, realice algunas prácticas sencillas con la librería de pygame, esto porque aun no estaba familiarizado con la biblioteca. Ya familiarizado con la biblioteca comencé a desarrollar el juego de la vida.

Algoritmo 1:

1. Crear una matriz de nxm
2. Llenar la de ceros

3. Introducir la posición de las células vivas, teniendo en cuenta que un cero dentro de la matriz representa una célula muerta y un uno una célula viva.
4. Recorrer la matriz evaluando las reglas del juego de la vida
5. Modificar la matriz con los nuevos valores ya habiendo aplicado las reglas del juego de la vida.
6. Repetir el proceso hasta que ya no se posible aplicar las reglas.

Observaciones.

- Fue relativamente fácil desarrollar el entorno gráfico en python y pygame.
- El programa se volvía lento con matrices mayores a 500 y tronaba con 1000.
- El algoritmo es lento porque debe recorrer matrices muy grandes, donde muchas veces evalúa valores innecesarios.

2.5 Segunda iteración

Teniendo en cuenta lo que paso en la primera iteración preste atención a la optimización del primer algoritmo, así que me senté a pensar un poco en como mejorar el algoritmo. El punto mas importante fue el recorrido de la matriz, ya que era esta parte la que hacia lento el programa, lo que hice fue seguir usando la matriz anterior, pero me apoye de una lista, en la cual guardaba las coordenadas de las células vivas, este arreglo mejore bastante el algoritmo porque al tener los valores de las células vivas en la lista, ya no era necesario recorrer toda la matriz ya que ya se sabia donde ir a evaluar una célula viva.

Algoritmo 2:

1. Crear una matriz de nxm
2. Llenar la de ceros
3. Crear una lista unidimensional
4. Introducir la posición de las células vivas, teniendo en cuenta que un cero dentro de la matriz representa una célula muerta y un uno una célula viva.
5. Introducir en la lista unidimensional solo los valores de las células vivas
6. Recorrer la lista para obtener la coordenada de la célula viva dentro de la matriz.
7. Aplicar las reglas del juego de la vida a las células vivas dentro de la matriz
8. Modificar la matriz con los nuevos valores ya habiendo aplicado las reglas del juego de la vida.
9. Repetir el proceso hasta que ya no se posible aplicar las reglas.

Observaciones.

- Solo se evalúan las células vivas, no se evalúan las células muertas, lo que provoca que el algoritmo sea mas rápido pero no haga lo que estamos buscando.
- El programa ya trabajaba con matrices de 5000x5000 células

2.6 Tercera iteración

Esta parte del desarrollo fue un poco pesada para mí, ya que no se me ocurría mucho para solucionar la parte de las células muertas, me canse un poco de pensar en una solución para el segundo algoritmo que incluso pensé en usar el primer algoritmo y usar un poco Cython para optimizar el código. Pero no estaba del todo convencido por lo que continué buscando soluciones, lo que se ocurrió fue que si ya tenía los valores de las células vivas podía usarlas para obtener las células muertas que eran candidatas a nacer. Una célula muerta que es candidata a nacer es una célula que está dentro del vecindario de la célula viva que se está analizando en un momento dado, entonces, cuando se encuentra una célula que es candidata lo que se hace es saltar a evaluar a esa célula con las reglas del juego de la vida. El algoritmo creció un poco sin embargo así era mejor que el primero ya que aun permitía procesar matrices de 5000x5000 células.

Algoritmo 3:

1. Crear una matriz de nxm
2. Llenar la de ceros
3. Crear una lista unidimensional
4. Introducir la posición de las células vivas, teniendo en cuenta que un cero dentro de la matriz representa una célula muerta y un uno una célula viva.
5. Introducir en la lista unidimensional solo los valores de las células vivas
6. Recorrer la lista para obtener la coordenada de la célula viva dentro de la matriz.
7. Al obtener las coordenadas de una célula viva, revisar en un vecindario si existen células muertas, si es así, entonces se debe aplicar las reglas del juego de la vida a cada célula muerta dentro del vecindario de la célula viva.
8. Aplicar las reglas del juego de la vida a las células vivas dentro de la matriz
9. Modificar la matriz con los nuevos valores ya habiendo aplicado las reglas del juego de la vida.
10. Repetir el proceso hasta que ya no se posible aplicar las reglas.

Observaciones.

- Con este algoritmo el sistema funciona de manera correcta además que ya se puede trabajar con matrices de 5000x5000 células, sin tener ningún problema.
- En esta parte de la implementación me llevé un poco de tiempo, por la solución del problema y la implementación en código, principalmente por problemas de lógica en el código y porque aun no dominaba muy bien pygame.

3 Funcionamiento de Simulador Game of Life

3.1 Interfaz para configuración inicial

La primer ventana que aparece cuando se inicia el programa nos permite darle la configuración inicial al juego de la vida. Nos permite insertar un tamaño de universo, insertar las reglas para la supervivencia y nacimiento de las células y por ultimo nos permite insertar un archivo de texto el cual ya contenga las coordenadas de las células vivas en un universo. Para iniciar el juego se deben insertar cada una de las configuraciones y dar clic en iniciar, cabe mencionar que la opción de abrir archivo es opcional.

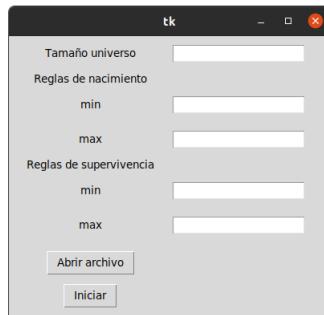


Figure 2: Interfaz para configuración inicial.

3.2 interfaz para cargar archivos

Esta interfaz nos permite navegar en la computadora donde se esta corriendo el juego, esto con la finalidad de seleccionar un archivo que ya contenga las coordenadas de las células vivas.

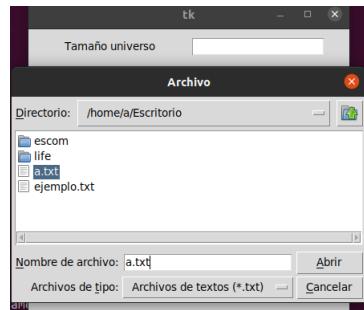


Figure 3: Interfaz para abrir una configuración de células del juego.

3.3 Interfaz con una configuración inicial

En la imagen se observa como ya se han introducido cada una de las opciones que pide el programa.



Figure 4: Interfaz para configuración inicial con una configuración inicial.

3.4 Pantalla principal de la simulación

Aquí ya se ve la interfaz del juego, para controlar el juego es importante conocer los controles.

- Pausa y reanudar juego - Flecha derecha
- Ver Gráfica de densidad- Flecha arriba
- Ver Gráfica de Entropía- Flecha abajo
- Guardar progreso - Flecha izquierda
- Botón izquierdo de mouse - Coloca una célula viva en la posición del puntero
- Botón derecho de mouse - Coloca una célula muerta en la posición del puntero
- Scroll del mouse - Zoom

Con esos sencillos controles ya se puede comenzar a jugar.

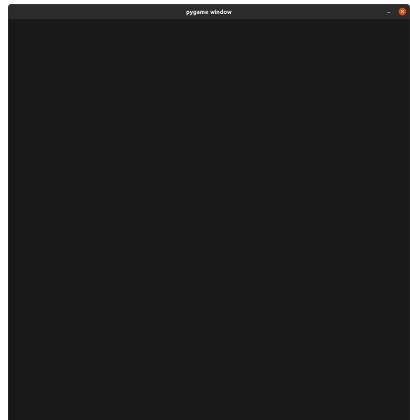


Figure 5: Interfaz gráfica del universo con células del juego.

3.5 Pantalla de simulación con células introducidas

En esta interfaz ya podemos ver como ya se hacen colocando unas cuantas células vivas en el universo del juego.

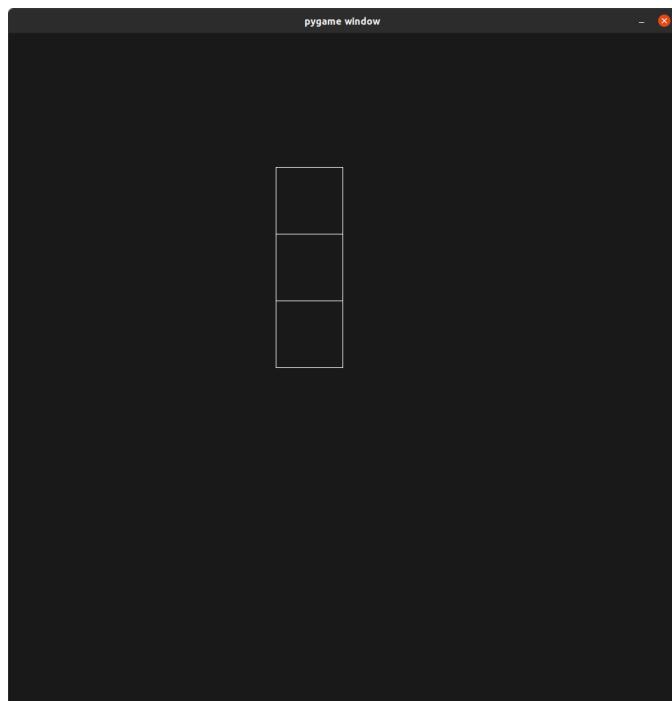


Figure 6: Interfaz gráfica del universo con células del juego.

3.6 Vista de simulación con células evolucionando

Ahora en esta interfaz se puede apreciar que se han colocado algunas células muertas y otras vivas.

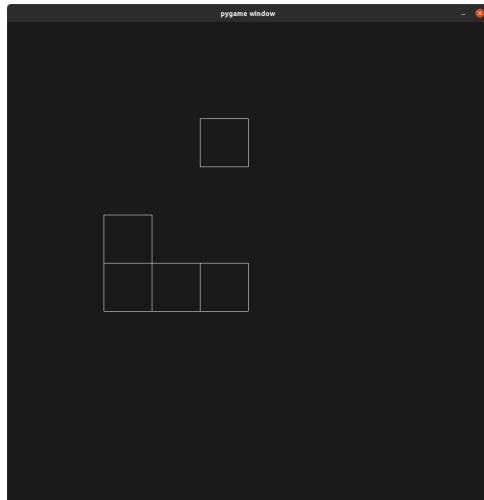


Figure 7: Interfaz gráfica del universo con células del juego.

3.7 Vista de la gráfica con densidad de células

En esta otra vista podemos ver la gráfica que nos permite ver el cambio del numero de células vivas en cada iteración.

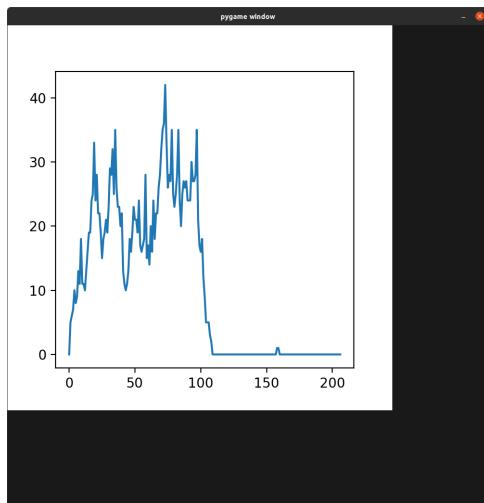


Figure 8: Interfaz de la gráfica de densidad.

3.8 Interfaz de guardado de configuración actual de la simulación

Finalmente si deseamos guardar todo lo que ha pasado en nuestro universo lo hacemos en la siguiente interfaz la cual nos da la opción nuevamente de navegar en nuestro computador para guardar nuestro progreso.

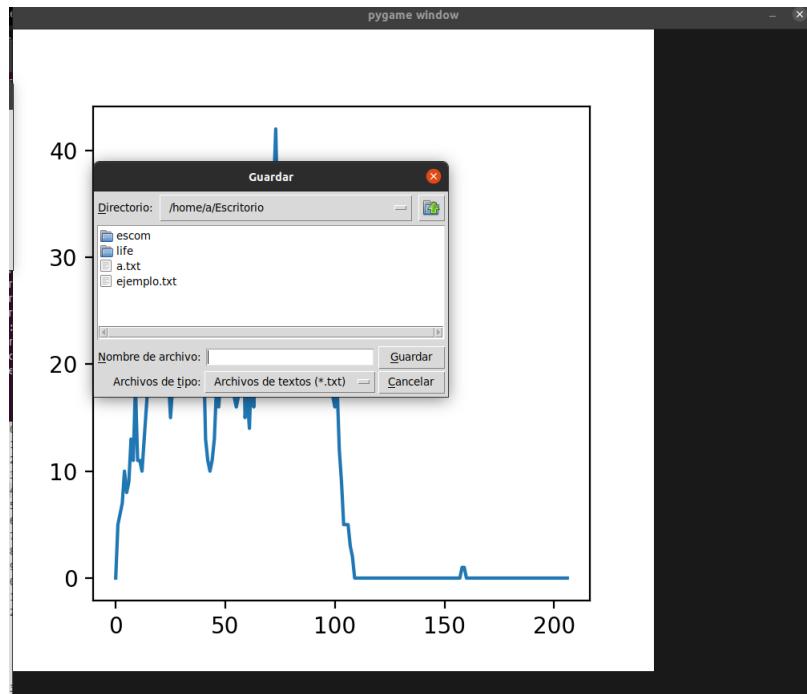


Figure 9: Interfaz para salvar configuración de células del juego.

3.9 Vista de la configuración de las células

Aquí podemos ver el archivo txt que el programa ha guardado, con las coordenadas de todas las células que se encontraban en el ultimo momento antes de guardar el progreso.

1	1 89
2	2 81
3	2 87
4	2 89
5	3 79
6	3 80
7	3 82
8	3 88
9	4 82
10	4 84
11	4 90
12	5 55
13	5 56
14	5 57
15	5 79
16	5 82
17	5 84
18	5 85
19	5 89
20	5 90
21	6 26
22	6 27
23	6 60
24	6 81
25	6 82
26	6 83
27	6 84
28	6 85
29	7 25
30	7 28
31	7 60
32	7 81
33	7 82
34	7 83
35	7 84
36	8 26
37	8 27
38	8 60
39	8 82
40	8 83
41	13 32
42	13 33
43	14 32
44	15 33
45	15 46
46	15 64
47	15 65
48	16 27
49	16 28
50	16 45
51	16 46

Figure 10: Archivo de texto con configuración de celular guardadas.

3.10 Pruebas con una matriz de 5000x5000

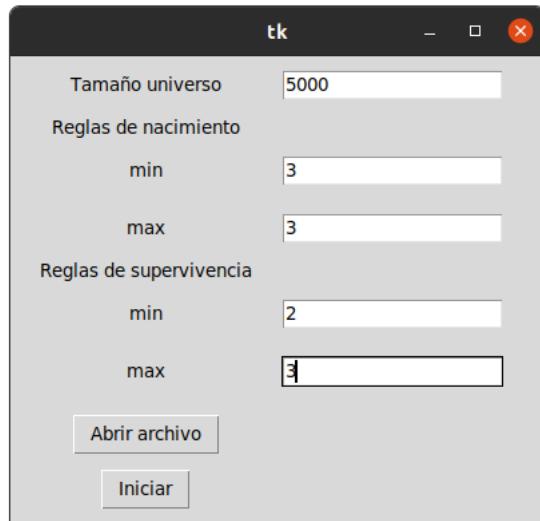


Figure 11: Se introduce la configuración para una matriz de 5000x5000 además de las reglas originales del juego de la vida.

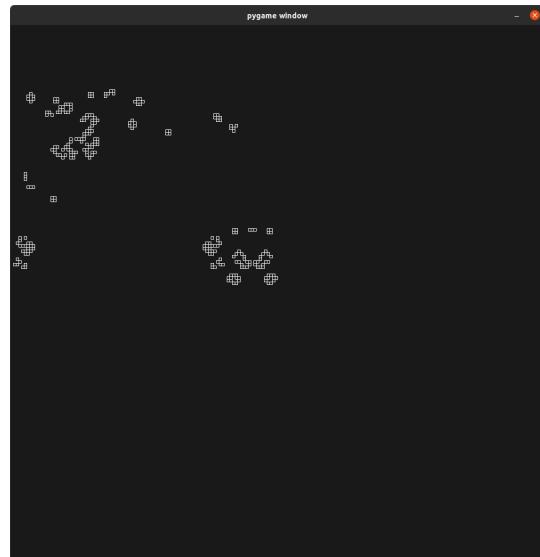


Figure 12: Se observa el comportamiento de las células en universo, vemos que el juego corre de manera fluida.

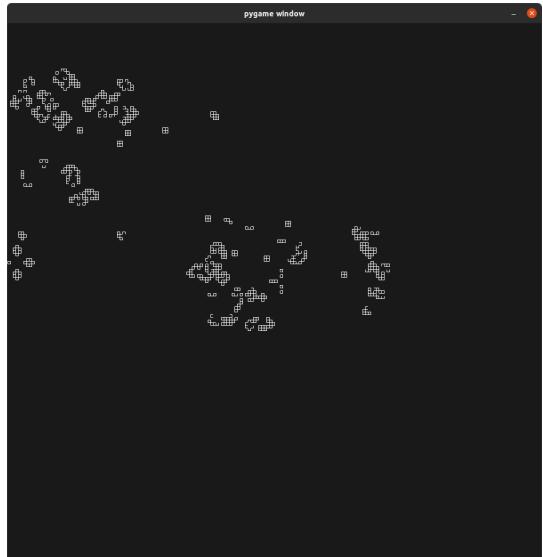


Figure 13: Movemos un poco la vista del universo para observar mejor los costados de del universo.

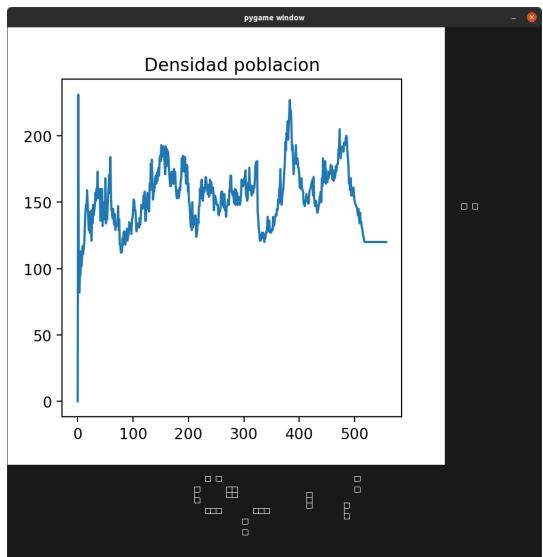


Figure 14: Se observa la gráfica donde vemos como ha ido cambiando el numero de células vivas.

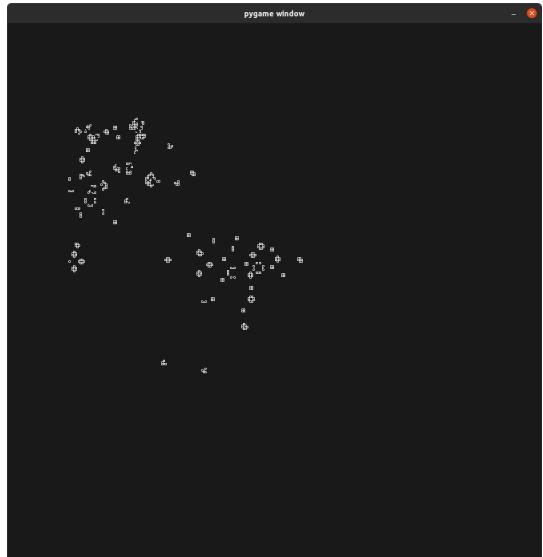


Figure 15: Volvemos a la vista del universo, nos alejamos un poco para ver como esta interactuando con el universo completo.



Figure 16: Nos alejamos un poco mas de las células vivas.

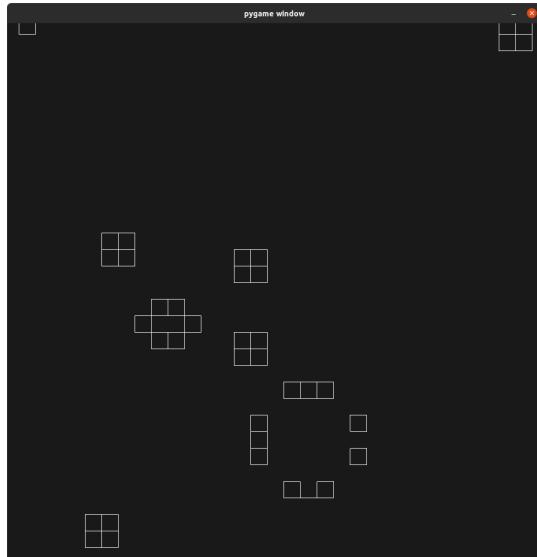


Figure 17: Nos acercamos a una sección del conjunto de células vivas y podemos observar algunas células cicladas.

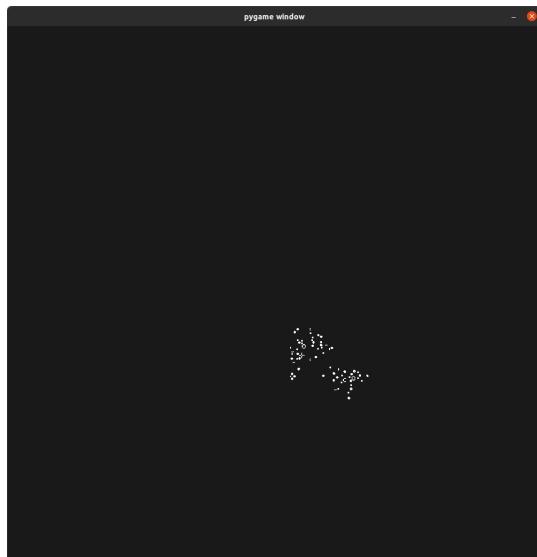


Figure 18: Finalmente volvemos a alejar el conjunto de las células vivas, y observamos que el programa se mantiene estable.

4 Modificación a la simulación

4.1 Interfaz de configuración inicial

La interfaz de la simulación fue modificada para integrarle la funcionalidad de calcular los atractores del universo. Además que se le modificó el formato en la que se insertan las reglas de evolución, ahora cuando se quiera ingresar una regla de evolución debe ser con el formato *B/S*.



Figure 19: Vista de interfaz principal de Game of Life y generador de atractores

4.2 Calculo de la Entropía

Para el calculo de la entropía se evaluó cada una de las vecindades del universo, conforme se iba revisando cada vecindad se iba registrando el numero de veces que apareció determina combinación o configuración. Ya con el valor de las repeticiones de cada configuración que aparecía se aplicó la siguiente formula.

Nota: Para observar la gráfica de Entropía se debe usar el botón de abajo del teclado y para observar la de densidades se usa la flecha de arriba.

$$\sum_{i=0}^{S-1} \left(\frac{Q_i^t}{n} * \log_2 \left(\frac{Q_i^t}{n} \right) \right)$$

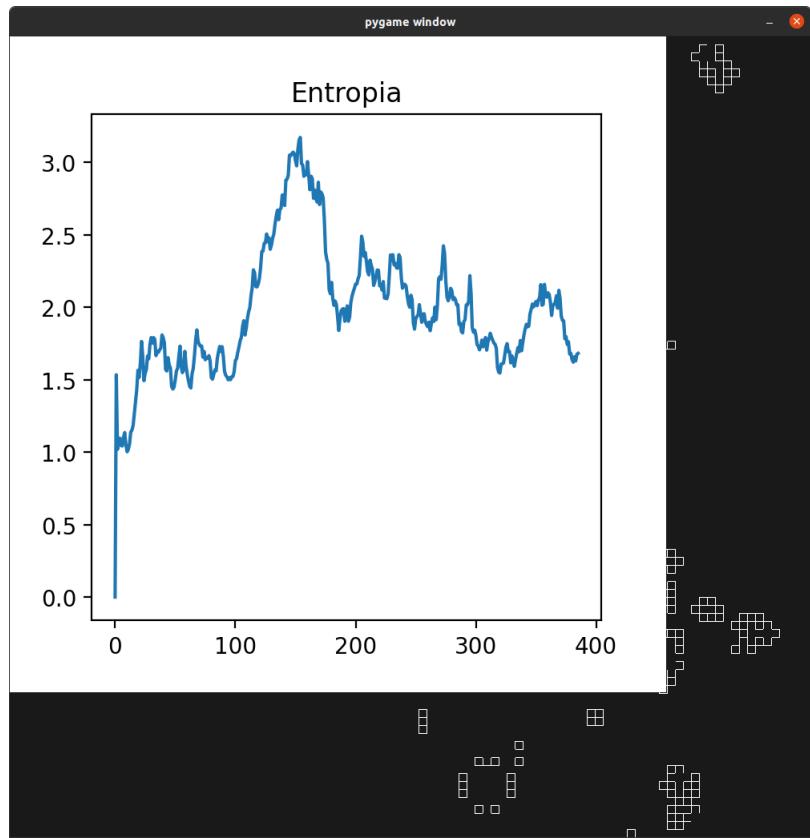


Figure 20: Vista 1 de la grafica de la Entropia.

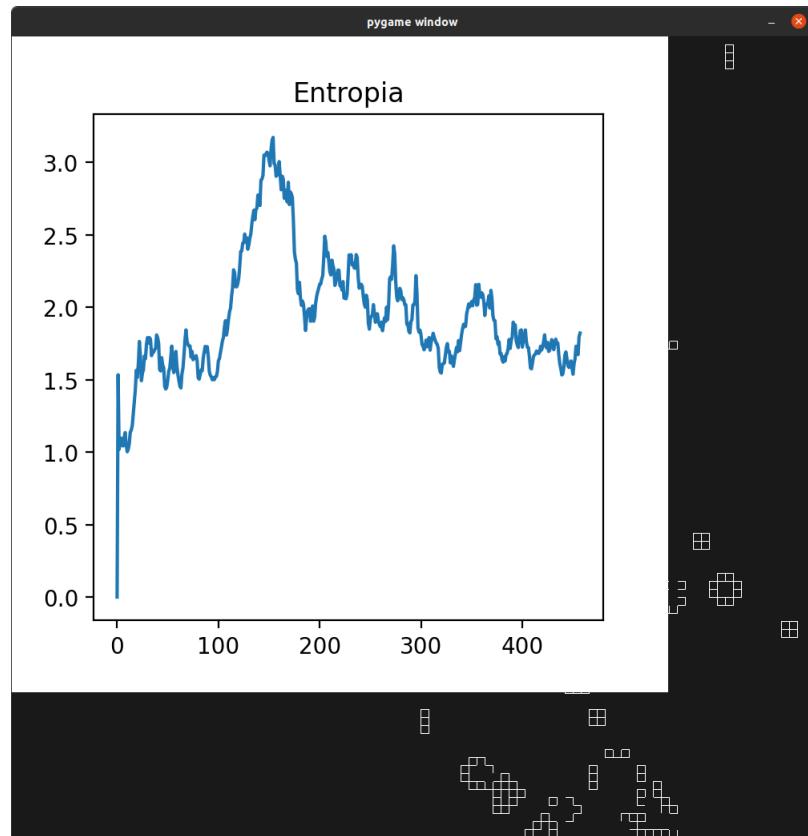


Figure 21: Vista 2 de la grafica de la Entropia.

4.3 Código de implementación de Game of life

```

1 import pygame, sys
2 import numpy as np
3 import time
4
5 import matplotlib.pyplot as plt
6 from matplotlib import pyplot
7 from matplotlib.animation import FuncAnimation
8
9 import matplotlib.animation as animation
10 from random import randrange
11 from datetime import datetime
12
13 #IMPORTAMOS TODA LA LIBRERIA
14 from tkinter import filedialog
15 import tkinter
16
17 import matplotlib
18 #matplotlib.use("Agg")
19
20 import matplotlib.backends.backend_agg as agg
21
22 import pylab

```

```

23
24 import pygame
25 from pygame.locals import *
26
27 xe = [0]
28 ye = [0]
29
30 yee = [0]
31
32 cadena = ""
33 ventan = tkinter.Tk()
34 v = tkinter.Entry(ventan)
35 supervivencial = tkinter.Entry(ventan)
36 nacimiento1 = tkinter.Entry(ventan)
37
38 supervivencia2 = tkinter.Entry(ventan)
39 nacimiento2 = tkinter.Entry(ventan)
40
41 estados = []
42 newestados = []
43 variable = 0
44 planox=0
45 planoy = 0
46
47 #CREAMOS LAS VENTANAS
48
49 def abrirArchivo():
50     global cadena
51     archivo = filedialog.askopenfilename(title="Archivo", filetypes=[("Archivos
      de textos", "*.txt")])
52     f = open(archivo)
53     cadena = f.read()
54
55 def funcion():
56     global xe
57     global ye
58     global v
59     global estados
60     global newestados
61     global variable
62     global planox
63     global planoy
64     global nacimiento1
65     global nacimiento2
66     global supervivencial
67     global supervivencia2
68
69     iteraciones = 0
70     vivas = 0
71     frecuencia = []
72     variable = (int)(v.get())
73     separado = (nacimiento1.get()).split("/")
74     min_nacimientos = (int)(separado[0][1])
75     max_nacimientos = (int)(separado[0][1])
76
77     if(len(separado[1]) == 2):
78         min_supervivencia =(int)(separado[1][1])
79         max_supervivencia =(int)(separado[1][1])
80     else:
81         min_supervivencia =(int)(separado[1][1])
82         max_supervivencia =(int)(separado[1][2])
83

```

```

84     ventan.destroy()
85
86     ##SE CREA LA PANTALLA
87     pygame.init()
88     width, height = 1000,1000
89
90     #SE CREA LA PANTALLA
91     screen = pygame.display.set_mode((height, width))
92
93     #COLOR DE LA PANTALLA
94     bg = 25, 25, 25
95
96     screen.fill(bg)
97
98     nxC, nyC = variable, variable
99
100    if variable >=1000:
101        dimCH = 5
102        dimCW = 5
103    else:
104        dimCW = (width / nxC)
105        dimCH = (height / nyC)
106
107    #ESTADO DE LA CELDA VIVA O MUERTA
108
109    gameState = np.zeros((nxC, nyC))
110
111    # print("tamnio",len(cadena))
112
113    aux = ""
114    for x in range(0,len(cadena)-1):
115        try:
116            entero = int(cadena[x])
117            #print(entero)
118            aux = aux + cadena[x]
119        except ValueError:
120            #print(int(aux), "termino de linea")
121            if cadena[x] == '\n':
122                gameState[posiX, int(aux)] = 1
123                estados.append((posiX, int(aux)))
124            else:
125                posiX = int(aux)
126            aux = ""
127
128    #control de flujo la ejecucion
129    pauseExect = True
130    grafica = False
131    grafica2 = False
132
133
134    #BUCLE DE EJECUCION
135    while True:
136
137        #SALIR DE LA PANTALLA
138        for event in pygame.event.get():
139            if event.type == pygame.QUIT:
140                sys.exit()
141
142        newGameState = np.copy(gameState)
143
144        #PINTAR LA PANTALLA
145        screen.fill(bg)

```

```

146
147 #ZONA DE DIBUJO
148 time.sleep(0.001)
149 ev = pygame.event.get()
150
151 for event in ev:
152     if event.type == pygame.KEYDOWN:
153         if event.key == pygame.K_LEFT:
154             pauseExect = not pauseExect
155             iteraciones = iteraciones - 1
156
157         if event.key == pygame.K_RIGHT:
158             pauseExect = not pauseExect
159             iteraciones = iteraciones - 1
160             guardado = filedialog.asksaveasfile(mode='w', title="Guardar",
161                                         confirmoverwrite=True, defaultextension=[('Archivos de textos','*.txt')],
162                                         filetypes=[('Archivos de textos','*.txt')])
163             if guardado is None:
164                 print("No se guardo")
165             for x in range(0, nxC):
166                 for y in range(0, nyC):
167                     if newGameState[x,y] == 1:
168                         texto = str(x) + " " + str(y) + '\n'
169                         guardado.write(texto)
170             guardado.close()
171
172         if event.key == pygame.K_UP:
173             grafica = not grafica
174             grafica2 = False
175
176         if event.key == pygame.K_DOWN:
177             grafica2 = not grafica2
178             grafica = False
179
180 mouseClick = pygame.mouse.get_pressed()
181
182 #DIBUJAR CELULAS
183 if sum(mouseClick) > 0:
184     posX, posY = pygame.mouse.get_pos()
185     celX, celY = int(np.floor(posX/ dimCW)), int(np.floor(posY/ dimCH))
186     if(newGameState[celX, celY] != 1):
187         estados.append((celX, celY))
188
189     newGameState[celX, celY] = not mouseClick[2]
190
191 if event.type == pygame.MOUSEBUTTONDOWN:
192     posX, posY = pygame.mouse.get_pos()
193     celX, celY = int(np.floor(posX/ dimCW)), int(np.floor(posY/ dimCH))
194
195     print(planox)
196     print(planoy)
197
198     if event.button == 4:
199         posX, posY = pygame.mouse.get_pos()
200         celX, celY = int(np.floor(posX/ dimCW)), int(np.floor(posY/ dimCH))
201
202         dimCW= dimCW + 1;
203         dimCH = dimCH + 1;
204         planox = planox - celX
205         planoy = planoy - celY

```

```

205
206     #variable = variable - 1
207     elif event.button == 5 and dimCW > 1 and dimCH > 1:
208         if (dimCH) * (variable) > width:
209
210             dimCW= dimCW - 1;
211             dimCH = dimCH - 1;
212
213             planox = planox + celX
214             planoy = planoy + celY
215             vivas = 0
216             a=0
217             b=0
218
219             newestados = estados.copy()
220
221             for x in range(0, len(estados)):
222                 a,b = estados[x]
223
224                 if newGameState[a,b] == 1:
225                     vivas = vivas + 1
226                 if not pauseExec:
227                     n_height = gameState[(a-1) % nxC, (b-1) % nyC] + \
228                         gameState[(a) % nxC, (b-1) % nyC] + \
229                         gameState[(a+1) % nxC, (b-1) % nyC] + \
230                         gameState[(a-1) % nxC, (b) % nyC] + \
231                         gameState[(a+1) % nxC, (b) % nyC] + \
232                         gameState[(a-1) % nxC, (b+1) % nyC] + \
233                         gameState[(a) % nxC, (b+1) % nyC] + \
234                         gameState[(a+1) % nxC, (b+1) % nyC]
235
236                     valorDecimal = 256 * gameState[(a-1) % nxC, (b-1) % nyC] + \
237                         128 * gameState[(a) % nxC, (b-1) % nyC] + \
238                         64 * gameState[(a+1) % nxC, (b-1) % nyC] + \
239                         32 * gameState[(a-1) % nxC, (b) % nyC] + \
240                         16 * gameState[(a) % nxC, (b) % nyC] + \
241                         8 * gameState[(a+1) % nxC, (b) % nyC] + \
242                         4 * gameState[(a-1) % nxC, (b+1) % nyC] + \
243                         2 * gameState[(a) % nxC, (b+1) % nyC] + \
244                         1 * gameState[(a+1) % nxC, (b+1) % nyC]
245                     frecuencia.append(int(valorDecimal))
246
247
248
249
250             #regla 2
251             if gameState[a,b] == 1 and (n_height < min_supervivencia or n_height >
252             max_supervivencia):
253                 newGameState[a,b] = 0
254                 newestados.remove(estados[x])
255
256             #regla 1
257
258             #Revisar vecino 1
259             if gameState[(a-1) % nxC,(b-1) % nyC] == 0:
260                 ceroX = (a-1) % nxC
261                 ceroY = (b-1) % nyC
262                 ceros = gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] + \
263                     gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
264                     gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
265                     gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
266                     gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \

```

```

266         gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
267         gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
268         gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
269
270     valorDecimal = 256 * gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] +
271     \
272         128 * gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
273         64 * gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
274         32 * gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
275         16 * gameState[(ceroX) % nxC, (ceroY) % nyC] + \
276         8 * gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
277         4 * gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
278         2 * gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
279         1 * gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
280
281     frecuencia.append(int(valorDecimal))
282
283     if ceros >= min_nacimientos and ceros <= max_nacimientos:
284         newGameState[ceroX,ceroY] = 1
285         newestados.append((ceroX,ceroY))
286
286     poly = [( ((ceroX) * dimCW) + planox,   (ceroY * dimCH) +
287     planoy),
287     ( ((ceroX+1) * dimCW) + planox,   (ceroY * dimCH) + planoy),
288     ( ((ceroX+1) * dimCW) + planox,   ((ceroY+1) * dimCH) + planoy)
289     ,
290     ( ((ceroX) * dimCW) + planox,   ((ceroY+1) * dimCH) + planoy)
291 ]
292
293
294
295 #Revisar vecino 2
296 if gameState[(a-1) % nxC ,(b+1) % nyC] == 0:
297     ceroX = (a-1) % nxC
298     ceroY = (b+1) % nyC
299     ceros = gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] + \
300         gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
301         gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
302         gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
303         gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
304         gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
305         gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
306         gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
307
308     valorDecimal = 256 * gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] +
309     \
310         128 * gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
311         64 * gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
312         32 * gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
313         16 * gameState[(ceroX) % nxC, (ceroY) % nyC] + \
314         8 * gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
315         4 * gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
316         2 * gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
317         1 * gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
318
319
320
321     frecuencia.append(int(valorDecimal))
322
323
324     if ceros >= min_nacimientos and ceros <= max_nacimientos:

```

```

323     newGameState[cerox,ceroY] = 1
324     newestados.append((cerox,ceroY))
325
326     poly = [((cerox) * dimCW) + planox, ((ceroY * dimCH) +
327     planoy),
328             ((cerox+1) * dimCW) + planox, ((ceroY * dimCH) + planoy),
329             ((cerox+1) * dimCW) + planox, ((ceroY+1) * dimCH) + planoy)
330
331             ((cerox) * dimCW) + planox, ((ceroY+1) * dimCH) + planoy)
332 ]
333
334     pygame.draw.polygon(screen, (255,255,255), poly, 1)
335
336     #Revisar vecino 3
337     if gameState[(a+1) % nxC,(b-1) % nyC] == 0:
338         ceroX = (a+1) % nxC
339         ceroY = (b-1) % nyC
340
341         ceros = gameState[(cerox-1) % nxC, (ceroY-1) % nyC] + \
342             gameState[(cerox) % nxC, (ceroY-1) % nyC] + \
343             gameState[(cerox+1) % nxC, (ceroY-1) % nyC] + \
344             gameState[(cerox-1) % nxC, (ceroY) % nyC] + \
345             gameState[(cerox+1) % nxC, (ceroY) % nyC] + \
346             gameState[(cerox-1) % nxC, (ceroY+1) % nyC] + \
347             gameState[(cerox+1) % nxC, (ceroY+1) % nyC] + \
348             gameState[(cerox+1) % nxC, (ceroY+1) % nyC]
349
350         valorDecimal = 256 * gameState[(cerox-1) % nxC, (ceroY-1) % nyC] +
351
352             128 * gameState[(cerox) % nxC, (ceroY-1) % nyC] + \
353             64 * gameState[(cerox+1) % nxC, (ceroY-1) % nyC] + \
354             32 * gameState[(cerox-1) % nxC, (ceroY) % nyC] + \
355             16 * gameState[(cerox) % nxC, (ceroY) % nyC] + \
356             8 * gameState[(cerox+1) % nxC, (ceroY) % nyC] + \
357             4 * gameState[(cerox-1) % nxC, (ceroY+1) % nyC] + \
358             2 * gameState[(cerox) % nxC, (ceroY+1) % nyC] + \
359             1 * gameState[(cerox+1) % nxC, (ceroY+1) % nyC]
360
361         frecuencia.append(int(valorDecimal))
362
363         if ceros >= min_nacimientos and ceros <= max_nacimientos:
364             newGameState[cerox,ceroY] = 1
365             newestados.append((cerox,ceroY))
366
367             poly = [((cerox) * dimCW) + planox, ((ceroY * dimCH) +
368             planoy),
369                     ((cerox+1) * dimCW) + planox, ((ceroY * dimCH) + planoy),
370                     ((cerox+1) * dimCW) + planox, ((ceroY+1) * dimCH) + planoy)
371
372             ((cerox) * dimCW) + planox, ((ceroY+1) * dimCH) + planoy)
373
374             #Revisar vecino 4
375             if gameState[(a+1) % nxC,(b+1) % nyC] == 0:
376                 ceroX = (a+1) % nxC
377                 ceroY = (b+1) % nyC
378
379                 ceros = gameState[(cerox-1) % nxC, (ceroY-1) % nyC] + \
380                     gameState[(cerox) % nxC, (ceroY-1) % nyC] + \
381                     gameState[(cerox+1) % nxC, (ceroY-1) % nyC] + \
382                     gameState[(cerox-1) % nxC, (ceroY) % nyC] + \

```

```

378         gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
379         gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
380         gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
381         gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
382
383     valorDecimal = 256 * gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] +
384     \
385     128 * gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
386     64 * gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
387     32 * gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
388     16 * gameState[(ceroX) % nxC, (ceroY) % nyC] + \
389     8 * gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
390     4 * gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
391     2 * gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
392     1 * gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
393
394     frecuencia.append(int(valorDecimal))
395
396     if ceros >= min_nacimientos and ceros <= max_nacimientos:
397         newGameState[ceroX,ceroY] = 1
398         newestados.append((ceroX,ceroY))
399
400         poly = [(
401             ((ceroX) * dimCW) + planox,   (ceroY * dimCH) +
402             planoy),
403             (
404                 ((ceroX+1) * dimCW) + planox,   (ceroY * dimCH) + planoy),
405                 (
406                     ((ceroX+1) * dimCW) + planox,   ((ceroY+1) * dimCH) + planoy)
407             ,
408             (
409                 ((ceroX) * dimCW) + planox,   ((ceroY+1) * dimCH) + planoy)
410         ]
411
412         pygame.draw.polygon(screen, (255,255,255), poly, 1)
413
414
415 #Revisar vecino 5
416 if gameState[(a) % nxC, (b-1) % nyC] == 0:
417     ceroX = (a) % nxC
418     ceroY = (b-1) % nyC
419     ceros = gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] + \
420         gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
421         gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
422         gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
423         gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
424         gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
425         gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
426         gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
427
428     valorDecimal = 256 * gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] +
429     \
430     128 * gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
431     64 * gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
432     32 * gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
433     16 * gameState[(ceroX) % nxC, (ceroY) % nyC] + \
434     8 * gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
435     4 * gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
436     2 * gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
437     1 * gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
438
439     frecuencia.append(int(valorDecimal))
440
441     if ceros >= min_nacimientos and ceros <= max_nacimientos:
442

```

```

435     newGameState[ceroX,ceroY] = 1
436     newestados.append((ceroX,ceroY))
437
438     poly = [(
439         ((ceroX) * dimCW) + planox,   (ceroY * dimCH) +
440         planoy),
440         (
441             ((ceroX+1) * dimCW) + planox,   (ceroY * dimCH) + planoy),
441             (
442                 ((ceroX+1) * dimCW) + planox,   ((ceroY+1) * dimCH) + planoy)
442     ,
443         (
444             ((ceroX) * dimCW) + planox,   ((ceroY+1) * dimCH) + planoy)
445     ]
446
447     pygame.draw.polygon(screen, (255,255,255), poly, 1)
448
449
450     #Revisar vecino 6
451     if gameState[(a-1) % nxC, (b) % nyC] == 0:
452         ceroX =(a-1) % nxC
453         ceroY =(b) % nyC
454         ceros = gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] + \
455             gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
456             gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
457             gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
458             gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
459             gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
460             gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
461             gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
462
463         valorDecimal = 256 * gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] +
464     \
465             128 * gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
466             64 * gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
467             32 * gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
468             16 * gameState[(ceroX) % nxC, (ceroY) % nyC] + \
469             8 * gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
470             4 * gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
471             2 * gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
472             1 * gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
473
474         frecuencia.append(int(valorDecimal))
475
476         if ceros >= min_nacimientos and ceros <= max_nacimientos:
477             newGameState[ceroX,ceroY] = 1
478             newestados.append((ceroX,ceroY))
479
480             poly = [(
481                 ((ceroX) * dimCW) + planox,   (ceroY * dimCH) +
482                 planoy),
482                 (
483                     ((ceroX+1) * dimCW) + planox,   (ceroY * dimCH) + planoy),
483                     (
484                         ((ceroX+1) * dimCW) + planox,   ((ceroY+1) * dimCH) + planoy)
484     ,
485                 (
486                     ((ceroX) * dimCW) + planox,   ((ceroY+1) * dimCH) + planoy)
487             ]
488
489             pygame.draw.polygon(screen, (255,255,255), poly, 1)
490
491     #Revisar vecino 7
492     if gameState[(a+1) % nxC, (b) % nyC] == 0:
493         ceroX = (a+1) % nxC
494         ceroY = (b) % nyC
495         ceros = gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] + \
496             gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
497             gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
498             gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \

```

```

490     gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
491     gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
492     gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
493     gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
494
495     valorDecimal = 256 * gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] +
496     \
497     128 * gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
498     64 * gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
499     32 * gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
500     16 * gameState[(ceroX) % nxC, (ceroY) % nyC] + \
501     8 * gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
502     4 * gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
503     2 * gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
504     1 * gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
505
506     frecuencia.append(int(valorDecimal))
507
508     if ceros >= min_nacimientos and ceros <= max_nacimientos:
509         newGameState[ceroX,ceroY] = 1
510         newestados.append((ceroX,ceroY))
511
512         poly = [((ceroX) * dimCW) + planox, ((ceroY * dimCH) +
513         planoy),
514             ((ceroX+1) * dimCW) + planox, ((ceroY * dimCH) + planoy),
515             ((ceroX+1) * dimCW) + planox, ((ceroY+1) * dimCH) + planoy)
516
517             ((ceroX) * dimCW) + planox, ((ceroY+1) * dimCH) + planoy]
518
519         pygame.draw.polygon(screen, (255,255,255), poly, 1)
520
521     #Revisar vecino 8
522     if gameState[(a) % nxC, (b+1) % nyC] == 0:
523         ceroX = (a) % nxC
524         ceroY = (b+1) % nyC
525         ceros = gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] + \
526             gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
527             gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
528             gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
529             gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
530             gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
531
532         valorDecimal = 256 * gameState[(ceroX-1) % nxC, (ceroY-1) % nyC] +
533         \
534         128 * gameState[(ceroX) % nxC, (ceroY-1) % nyC] + \
535         64 * gameState[(ceroX+1) % nxC, (ceroY-1) % nyC] + \
536         32 * gameState[(ceroX-1) % nxC, (ceroY) % nyC] + \
537         16 * gameState[(ceroX) % nxC, (ceroY) % nyC] + \
538         8 * gameState[(ceroX+1) % nxC, (ceroY) % nyC] + \
539         4 * gameState[(ceroX-1) % nxC, (ceroY+1) % nyC] + \
540         2 * gameState[(ceroX) % nxC, (ceroY+1) % nyC] + \
541         1 * gameState[(ceroX+1) % nxC, (ceroY+1) % nyC]
542
543         frecuencia.append(int(valorDecimal))
544
545         if ceros >= min_nacimientos and ceros <= max_nacimientos:
546             newGameState[ceroX,ceroY] = 1
547             newestados.append((ceroX,ceroY))

```

```

547
548     poly = [((ceroX) * dimCW) + planox, ((ceroY * dimCH) +
549     planoy),
550             ((ceroX+1) * dimCW) + planox, ((ceroY * dimCH) + planoy),
551             ((ceroX+1) * dimCW) + planox, ((ceroY+1) * dimCH) + planoy)
552         ,
553             ((ceroX) * dimCW) + planox, ((ceroY+1) * dimCH) + planoy)
554 ]
555
556     pygame.draw.polygon(screen, (255,255,255), poly, 1)
557
558 #DIBUJAMOS EL CUADRADO
559 poly = [( (a) *dimCW) + planox, (b * dimCH) + planoy,
560           ((a+1) * dimCW) + planox, (b * dimCH) + planoy),
561           ((a+1) * dimCW) + planox, ((b+1) * dimCH) + planoy),
562           ((a) * dimCW) + planox, ((b+1) * dimCH) + planoy]
563
564 if newGameState[a,b] == 0:
565     pygame.draw.polygon(screen, (25,25,25), poly, 1)
566 else:
567     pygame.draw.polygon(screen, (255,255,255), poly, 1)
568
569 frecuenciaValores = collections.Counter(frecuencia)
570
571 entropia = 0
572
573 coeficiente = ((variable*variable) - (len(estados) * 9))/(variable*
574 variable)
575 entropia = entropia + ( coeficiente * math.log(1/coeficiente, 2) )
576 for llave, value in frecuenciaValores.items():
577     #print(llave , " repeticiones ", value)
578     coeficiente = value/(variable*variable)
579
580     entropia = entropia + ( coeficiente * math.log(1/coeficiente, 2) )
581
582 frecuencia.clear()
583
584 newestados = list(set(newestados))
585
586 estados = newestados.copy()
587
588 if pauseExect == False:
589     ye.append(vivas)
590     yee.append(entropia)
591     #print(ye)
592
593 #GRAFICA
594 if grafica == True:
595     fig = pylab.figure(figsize=[4.1, 4.1],
596                         dpi=200,
597                         )
598
599     ax = fig.gca()
600     ax.set_title('Densidad poblacion')
601     ax.plot(ye)
602
603     canvas = agg.FigureCanvasAgg(fig)
604     canvas.draw()
605     renderer = canvas.get_renderer()
606     raw_data = renderer.tostring_rgb()

```

```

605     pygame.init()
606
607     size = canvas.get_width_height()
608
609     surf = pygame.image.fromstring(raw_data, size, "RGB")
610     screen.blit(surf, (0,0))
611
612
613 #GRAFICA
614 if grafica2 == True:
615     fig = pylab.figure(figsize=[4.1, 4.1],
616                         dpi=200,
617                         )
618     ax = fig.gca()
619     ax.set_title('Entropia')
620     ax.plot(yee)
621
622     canvas = agg.FigureCanvasAgg(fig)
623     canvas.draw()
624     renderer = canvas.get_renderer()
625     raw_data = renderer.tostring_rgb()
626
627     pygame.init()
628
629     size = canvas.get_width_height()
630
631     surf = pygame.image.fromstring(raw_data, size, "RGB")
632     screen.blit(surf, (0,0))
633
634     gameState = np.copy(newGameState)
635
636 #ACTUALIZAR PANTALLA
637     pygame.display.flip()
638
639
640 def ventana():
641     global ventan,v
642     global nacimiento1
643     global nacimiento2
644     global supervivencial
645     global supervivencia2
646
647     ventan.geometry("400x230")
648
649     variable = 0
650
651     tamano = tkinter.Label(ventan, text ="Tama o universo")
652     tamano.grid(row=0, column=0, padx=20)
653
654     v.grid(row=0, column=1, pady=10)
655     ventan.title("Game of life")
656
657 ##REGLA DE NACIMIENTO
658     mas_nacido = tkinter.Label(ventan, text ="Reglas de nacimiento")
659     mas_nacido.grid(row=1, column=0, padx=20)
660
661     mas_nacido = tkinter.Label(ventan, text ="Formato B#/S#")
662     mas_nacido.grid(row=2, column=0, padx=20)
663
664     nacimiento1.grid(row=2, column=1, pady=10)
665

```

```

666     guardar = tkinter.Button(ventan, text="Abrir archivo", command=abrirArchivo,
667         background="#ABFDC5")
668     guardar.grid(row=7, column=0 , pady=10)
669
670     b = tkinter.Button(ventan, text ="Iniciar simulacion", width=0, height=0,
671         command=funcion, background="#ABFDC5")
672     b.grid(row=8, column=0)
673
674     b = tkinter.Button(ventan, text ="Atractores", width=0, height=0, command=
675         createGraph, padx=50,pady=15, background="#F67171")
676     b.grid(row=7, column=1, pady=10)
677
678     ventan.mainloop()
679
680 ventana()

```

5 Conclusiones

Lo primero que puedo mencionar es que ha sido una proyecto muy interesante de desarrollar, principalmente por las adecuaciones que se tuvieron que realizar el procesamiento de grandes cantidades de células, ya que como lo mencione en una parte del reporte estuve a punto de optar por una opción que me permitiera usar un algoritmo sencillo pero elevado en complejidad computacional, pero con un poco de paciencia logre adecuar el algoritmo original para que me permitiera correr mi programa sin problemas y con una matriz bastante grande.

Durante el desarrollo del proyecto logre aprender un poco mas de algunas tecnologías que desconocía, no voy a mentir al principio me resulto un poco complicado el abstraer ciertos concepto y entender algunos otros aun así logre seguir avanzando, debo mencionar que al terminar este proyecto me di cuenta que a veces es importante buscar mas de una solución para un problema, ya que el mas fácil puede ser el que de mas problemas.

Por otra parte he quedado realmente sorprendido del juego, es impresionante como con unas cuantas reglas se pueden construir "universos" los cuales pueden expandirse, sobrevivir y morir. Este juego tan pequeño me ha dejado pensando un poco mas sobre la existencia y mas aun cuando vemos algunas de sus configuraciones que a simple vista se ven sencillas pero esas configuraciones tan sencillas son capaces de generar una gran cantidad de células vivas que interactúan en el universo y entre si, lo que te hace darte cuenta de lo impresionante que resulta la existencia y mas aun la "suerte" para que todo aparezca.

6 Referencias

1. colaboradores de Wikipedia. (2020, 14 octubre). Juego de la vida. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Juego_de_la_vida
2. pygame - Empezando con pygame | pygame Tutorial. (2018, 1 octubre). Rip Tutorial. <https://riptutorial.com/es/pygame>
3. Matplotlib - Matplotlib | Matplotlib: Visualization with Python. (2012, 10 febrero). Matplotlib. <https://matplotlib.org/>
4. pygame - pygame | Pygame Front Page. (2015, 2 septiembre). Matplotlib. <https://www.pygame.org/docs/>

7 Introducción de Atractores

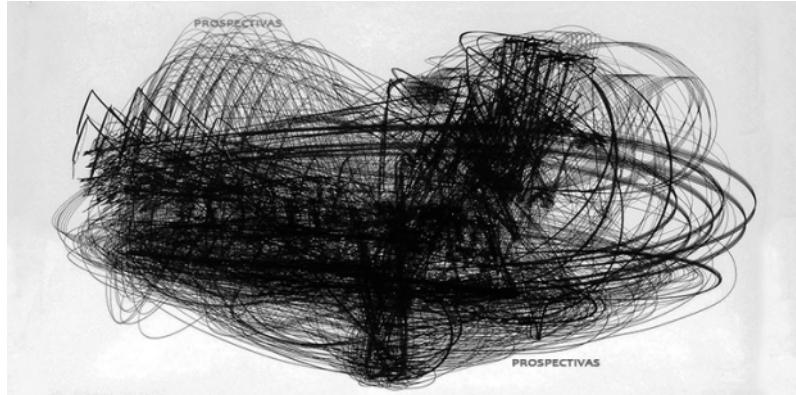


Figure 22: Atractor extraño

En los sistemas dinámicos, un atractor es un conjunto de valores numéricos hacia los cuales un sistema tiende a evolucionar, dada una gran variedad de condiciones iniciales en el sistema. Para que un conjunto sea un atractor, las trayectorias que le sean suficientemente próximas han de permanecer próximas incluso si son ligeramente perturbadas. Geométricamente, un atractor puede ser un punto, una curva, una variedad o incluso un conjunto complicado de estructura fractal conocido como atractor extraño. La descripción de atractores de sistemas dinámicos caóticos ha sido uno de los grandes logros de la teoría del caos.

La trayectoria del sistema dinámico en el atractor no tiene que satisfacer ninguna propiedad especial excepto la de permanecer en el atractor; puede ser periódica, caótica o de cualquier otro tipo.

8 Desarrollo

8.1 El juego

Se trata de un juego de cero jugadores, lo que quiere decir que su evolución está determinada por el estado inicial y no necesita ninguna entrada de datos posterior. El "tablero de juego" es una malla plana formada por cuadrados (las "células") que se extiende por el infinito en todas las direcciones. Por tanto, cada célula tiene 8 células "vecinas", que son las que están próximas a ella, incluidas las diagonales. Las células tienen dos estados: están "vivas" o "muertas" (o "encendidas" y "apagadas"). El estado de las células evoluciona a lo largo de unidades de tiempo discretas (se podría decir que por turnos). El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente.

Todas las células se actualizan simultáneamente en cada turno, siguiendo estas reglas:

1. Una célula muerta con exactamente 3 células vecinas vivas "nace" (es decir, al turno siguiente estará viva).

- Una célula viva con 2 o 3 células vecinas vivas sigue viva, en otro caso muere (por "soledad" o "superpoblación").

Nota. Para la realización de este proyecto se utilizaron

8.2 Creación de atractores usando el juego de la vida

8.3 Tecnologías utilizadas para el desarrollo del juego de la vida

- Python. Como lenguaje de programación para desarrollo con simplicidad, versatilidad y rapidez.
- NetworkX. Es una biblioteca de Python para el estudio de grafos y análisis de redes. NetworkX es un software libre publicado bajo la licencia BSD-new.
- Matplot. Permitió el desarrollo de gráficas en tiempo real, al ser una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

8.4 Primera iteración

En esta primera parte del desarrollo comencé conociendo un poco el funcionamiento de la librería Networkx, no conocía nada de la librería ya que esta era mi primera vez usándola. Ya con el conocimiento de la librería lo primero que hice fue reutilizar la función de Life para realizar las iteraciones sobre las diferentes configuraciones que se iban a generar mas adelante.

```

1 def functionLife(lenList, combinationsList):
2
3     newCombinationsList = copyListOfLists(combinationsList)
4
5     for x in range(0,lenList ):
6         valuesX = []
7         valuesY = []
8         for y in range(0,lenList):
9             livingCells = 0
10
11         valuesX = [
12             ((x-1) % lenList , (y-1) % lenList),
13             ((x) % lenList, (y-1) % lenList),
14             ((x+1) % lenList, (y-1) % lenList),
15             ((x-1) % lenList, (y) % lenList),
16             ((x+1) % lenList, (y) % lenList),
17             ((x-1) % lenList, (y+1) % lenList),
18             ((x) % lenList, (y+1) % lenList),
19             ((x+1) % lenList,(y+1) % lenList)
20         ]
21
22         valuesX = set(valuesX)
23
24         for index in valuesX:
25             a,b = index
26             livingCells = livingCells + int(combinationsList[a][b])
27

```

```

28     #print("livingCells ", livingCells)
29
30     #BORN
31     if ((combinationsList[x][y]) == 0 and (livingCells == 2)):
32         newCombinationsList[x][y] = 1
33     #SURVIVE
34     elif((combinationsList[x][y] == 1) and (livingCells == 7)):
35         newCombinationsList[x][y] = 0
36
return newCombinationsList

```

Para la función Life no se hizo ningún cambio significativo, se mantuvo igual a la que se utilizó en la simulación del juego de la vida.

Para empezar desarrolle una función que generara todas la combinaciones para cada universo de evoluciones que se desee. La idea generar es que dado un universo. Por ejemplo 2×2 , las combinaciones posibles son 2^4 esto se debe a que el universo de evoluciones es una matriz con filas y columnas, pero podemos representar esa matriz en un vector el cual sera de 1×4 el 4 es porque tenemos 2 filas de 2 columnas. Ya conociendo la longitud del vector ya solo queda hacer todas las combinaciones para un vector de longitud 4 con la posibilidad de poner 0 o 1 en cada posición del vector. Ya con esto en mente podemos decir que para un universo de $n \times n$ el numero de combinaciones posibles seria 2^n lo que nos indica que este problema crece exponencialmente.

Se creo una función que obtiene todas las combinaciones posibles de un universo en específico.

```

1 def getCombinations(lenList, combination):
2     global combinationsList;
3     newCombination = ""
4
5     for x in range(0,2):
6         if(lenList != 1):
7             newCombination = combination + str(x)
8             getCombinations(lenList-1, newCombination)
9             newCombination = ""
10        else:
11            newCombination = combination + str(x)
12            combinationsList.append(newCombination)
13            newCombination = ""
14    combination = ""

```

Para la función usa una lista que almacena cada combinación obtenida, la función inicia poniendo en 0 el bit mas significativo hasta el bit menos significativo, cuando se lleva la bit menos significativo se guarda la combinación que se ha obtenido hasta el momento y se cambia el estado del bit menos significativo a 1, esta es otra combinación así que se guarda cuando ya se pusieron en el bit menos significativo todas sus posibilidades de valores(0 y 1) entonces se retrocede de la función recursiva y se cambia a 1 el bit que antecede al menos significativo y se vuelve a entrar a la función recursiva para obtener las otras posibilidades del bit menos significativa y este proceso se repite hasta llegar al bit mas significativo de todo el vector.

Ya con todas las combinaciones posibles lo siguiente fue colocar todas esas combinaciones dentro de una matriz, esta matriz es la configuración dentro del universo $n \times n$.

```

1 def stringToMatriz(combination):
2     column = []

```

```

3  row = []
4  STOP = int(math.sqrt(lenList))
5  index = 0
6
7  for x in range(0,len(combination)):
8      for y in range(0,len(combination[x])):
9
10     if(index < STOP):
11         row.append(int(combination[x][y]))
12
13     if( index == STOP -1):
14         column.append(row.copy())
15         row.clear()
16         index = 0
17     else:
18         index += 1
19
20 cases.append(column.copy())
21 column.clear()
22 return cases

```

Esta función lo único que hace es crear una matriz e insertar filas del tamaño de la raíz cuadrada del exponente de las combinaciones , por ejemplo 2^4 aquí la raíz cuadrada de 4 es 2 por lo que el tamaño de las filas serán de longitud 2. La razón por la se decidió hacer esto fue por el código de la función Life recibe como parámetro una matriz. Además que ya con los datos en la matriz es mas sencillo visualizar las configuraciones que se obtienen después de aplicar la regla de Life.

Con esta función se concluye la primera iteración, hasta este punto ya tengo desarrollado todos las configuración y tengo la función que evalúa cada una de ellas, el siguiente paso fue evaluar cada configuración y definir una condición de parada pero eso lo vemos en la siguiente iteración. Hasta el momento el algoritmo nos quedo de la siguiente manera:

Algoritmo 1:

1. Generar todas las posibles combinaciones
2. Crear matrices con cada una de las combinaciones y almacenarlas en una lista de configuraciones de matrices.

Observaciones.

- Hasta este punto me costo un poco de trabajo realizar la función para generar todas las combinaciones, esto porque al principio tenia la idea de como implementarlo pero en el desarrollo de la función surgieron varios problemas debido a la condición de parada. Para solucionar el problema tuvo que hacer algunas pruebas de escritorio que me hicieran ver y entender de mejor manera el problema pero al final se concluyo con éxito.
- Hasta este punto no se noto problemas de procesamiento o lentitud del programa por lo que se decidió continuar.

8.5 Segunda iteración

En esta iteración se tiene el objetivo de generar un grafo usando las configuraciones obtenidas en la iteración pasada y las configuraciones que se obtienen al aplicar la regla de Life a esas mismas configuraciones. Entonces lo que hice es crear una función que evalúa cada una de las configuraciones obtenidas.

En la función para generar el grafo lo primero que debemos hacer es generar las combinaciones de algún universo de evoluciones de $n \times n$ con las combinaciones listas se deben crear las configuraciones de matrices ya que serán estas las que se evaluarán en la función del grafo.

```
getCombinations(lenList, globalCombination )  
  
print("Generated combination ",len(combinationsList))  
  
stringToMatriz(combinationsList)
```

Los datos obtenidos por la función `CreateGraph()` son almacenados en 2 archivos de texto uno que guarda los vértices del grafo y otro que guarda las aristas del grafo.

```
1 path = "nodes.txt"  
2 n = open(path, 'w')  
3  
4 path = "edges.txt"  
5 e = open(path, 'w')
```

A continuación se muestra la función que crea el grafo. Lo primero que hace la función es crear una lista llamada *Graph*, esta lista almacena los vértices que se van generando en cada iteración. También se declara una variable llamada *decimal*, que almacenará la matriz que se está evaluando por la función.

```
1 graph = []  
2 G = nx.Graph()  
3 decimal = 0
```

Después tenemos un ciclo `for`, este ciclo hace un recorrido de todas las combinaciones que se generaron con la función `getCombinations`, la variable *decimal* toma el valor de la combinación que se está evaluando en el momento, la variable *variableFunctionLife* almacena la configuración que se consigue cuando se aplica la función `life`, pero como ésta es la primera iteración en la variable se encuentra la misma configuración que en la variable *decimal*. Finalmente antes de entrar al `while` se declaran 3 variables más *iteracion* para llevar el valor de la

```
1 for x in range(0, len(cases)):  
2     decimal = cases.index(cases[x])  
3     variableFunctionLife = cases[x].copy()  
4     iteracion = 0  
5     antecesor = 0  
6     sem = 0  
7     ...
```

Para explicar esta parte de la función comenzaremos definiendo algunas de las variables y que papel desempeñan dentro de la función.

- actual: Esta es el valor en decimal de la combinación que se está evaluando.
- antecesor: Este es el valor en decimal del ancestro de actual. Esta variable nos ayuda a saber si en alguna iteración de una combinación se generó un ancestro y se ancestro le corresponde a la combinación actual que se está evaluando.
- G: Es el grafo que almacena vértices y aristas.
- iteración: Lleva el conteo de las iteraciones que se le ha hecho a alguna combinación. La usamos para identificar cuando una determinada combinación tiene o no tiene antecesores.
- variableFunctionLife: Esta es la variable de la combinación que se genera cuando se aplica la regla Life sobre la variable actual.

Ya el conocimiento de cada variable explicaremos que hace el while, al inicio del while tenemos una condición la cual pregunta si el valor actual que se está evaluando ya existe dentro del grafo.

Cuando no existe:

1. Aumentamos en un la variable iteración
2. Registramos el vértice en el grafo.
3. Verificamos si ese vértice tiene antecesor solo si tiene antecesor se lo registramos.
4. Asignamos el valor de actual a antecesor
5. Aplicamos la función Life a la variableFunctionLife que tiene el valor de la configuración actual.
6. Guardamos en la variable actual el valor en decimal que se ha generado al aplicar la regla de Life a la combinación actual.
7. Se repite el ciclo while

Cuando existe:

1. Se pregunta si ya se ha iteración es igual a cero, porque si es igual a cero significa que estamos evaluando una de las combinaciones de las que se generaron antes pero algún otro vértice ya llegó a esa combinación a través de otra combinación.
2. Por otro lado si el nodo ya existe en el grafo y además su iteración es mayor a cero entonces eso significa que se ha encontrado un ciclo por lo que se guarda el vértice que se está evaluando junto con su ancestro y se rompe el ciclo while para seguir revisando el resto de combinaciones que se generaron antes.

```

1  while(True):
2      if(actual in G.nodes):
3          if(iteracion == 0):
4              G.add_node(actual)
5              n.write(str(actual))
6              antecesor=actual
7              variableFunctionLife = functionLife(len(variableFunctionLife),
variableFunctionLife)
8              decimal = cases.index(variableFunctionLife)
9              iteracion+=1
10         else:
11             #print(" equal ")
12             G.add_edge(actual, antecesor)
13             union = str(actual) + " "+str(antecesor) + "\n"
14
15             e.write(union)
16             break
17         else:
18             iteracion+=1
19             G.add_node(actual)
20
21             n.write(str(actual)+ "\n")
22
23         if(iteracion != 1):
24             union = str(actual) + " " + str(antecesor) + "\n"
25             e.write(union)
26             G.add_edge(actual,antecesor)
27
28         antecesor=actual
29         variableFunctionLife = functionLife(len(variableFunctionLife),
variableFunctionLife)
30         actual = cases.index(variableFunctionLife)

```

Algoritmo 2:

1. Generar todas las posibles combinaciones
2. Crear matrices con cada una de las combinaciones y almacenarlas en una lista de configuraciones de matrices.
3. Generamos el grafo usando las combinaciones generadas anteriormente.

Observaciones.

- Esta fue una de las funciones que mas tiempo me llevo ya que surgieron bastantes problemas y algunas dudas, principalmente en la forma en la que debía generarse el grafo, para resolver esas dudas tuve que recurrir a un vídeo de la clase donde se explicaba un poco mas de la generación de ciclos. Al final opte por realizar algunos ejemplos en papel de lo que se me solicitaba y fue así como termine de entender lo que se solicitaba además que con esos ejemplos en papel logre resolver mas rápido los problemas que iban apareciendo al momento de desarrollar la función.

8.6 Tercera iteración

Hasta este punto el programa desarrollado ya es capaz de generar todas la combinaciones de universos de $n \times n$, también es capaz de evaluar cada una de

esas combinaciones para generar un grafo. Este grafo tiene la característica de conectar un vértice(padre) con otro vértice(hijo) el vértice padre al vértice padre se le aplica la función Life y da como resultado el vértice hijo, con esta lógica fue como se fue armando el grafo, el grafo se termina de armar cuando la ultima combinación ha sido evaluada.

Una vez tenemos le grafo ya armado lo que falta por hacer es graficarlo, para graficarlo se utilizo la librería Networkx de python.

Para terminar lo que se hace es identificar el numero de ciclos que aparecen en cada universo con determina regla de evolución. Para realizar esa tarea nos auxiliamos de una lista llamada *ciclos*, a esta lista se le registra un nodo al principio de la evaluación de alguna combinación y también se le registra un valor cada que se le aplica la regla de Life a dicha combinación, al final cuando se encuentre un ciclo, la lista *ciclos* tendrá un numero determinado de vértices los cuales indicaran si se ha encontrado un ciclo o no, para saber si ha encontrado un ciclo basta con revisar el primer y el ultimo valor de la lista, si son iguales entonces ahí aparecido un ciclo, ya que se encuentra el ciclo se limpia la lista y se continua evaluando las combinaciones restantes.

Algoritmo generado:

1. Generar todas las posibles combinaciones
2. Crear matrices con cada una de las combinaciones y almacenarlas en una lista de configuraciones de matrices.
3. Verificar si al final de la evaluación de la combinación se obtuvo un ciclo, si se obtuvo se aumenta el numero de ciclos.
4. Generamos el grafo usando las combinaciones generadas anteriormente.
5. Introducir el grafo en la función draw de Networkx y esperar que se grafique.
6. Mostrar el numero de ciclos obtenidos en el sistema.

Observaciones.

- En esta parte del desarrollo el principal problema al que me enfrenté fue el aprender a utilizar la librería Networkx, ya que cuando comencé los grafos no se entendían muy bien además que en ocasiones ni siquiera se generaban por lo que tuve que documentarme y experimentar con la librería.
- Para le conteo de los ciclos tuve que volver a observar el comportamiento del sistema para encontrar de nuevo la mejor manera de llevar el conteo de los ciclos, al principio pensé en solo poner una variable que contara ciertos casos especiales de ciclado pero no resultó buena idea al final decidí en usar la lista para llevar el conteo.

9 Funcionamiento

Al final el código que se desarrollo se integro a la simulación del universo de $n \times n$, quedando de la siguiente manera.

En la vista principal podemos observar una interfaz similar a la primera versión de la simulación del juego de la vida, pero ahora ya le ha integrado la opción de generar atractores de un universo y especificando las reglas de ese universo, las reglas tienen el formato B/S .

9.1 Ventana principal de configuración de simulador de Game of life y generador de atractores



Figure 23: Vista de interfaz principal de Game of Life y generador de atractores

9.2 Ventana de visualización de grafo

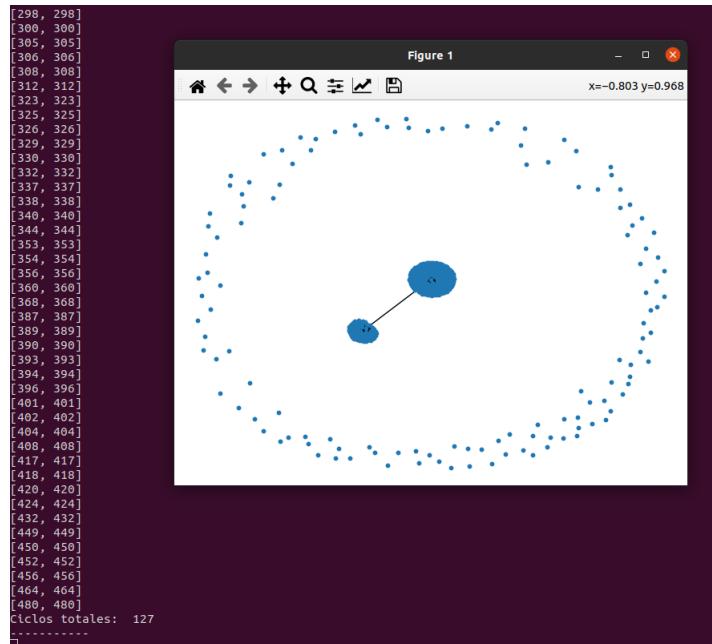


Figure 24: Vista de grafo 3x3 regla B3/S23 y numero de ciclos encontrados

Al final de la ejecución del generador de atractores de generan 2 archivos de texto, uno que almacena todos los nodos generados y otro que guarda las aristas generadas.

9.3 Vista de archivos de texto generados por el generador de atractores

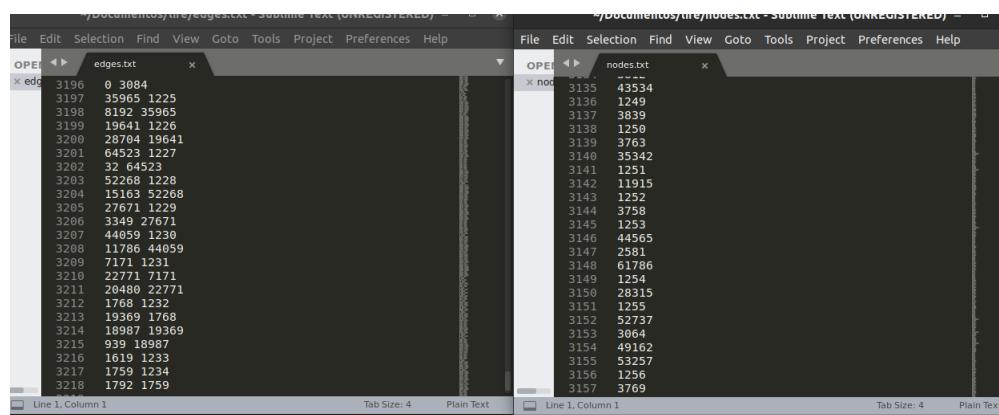
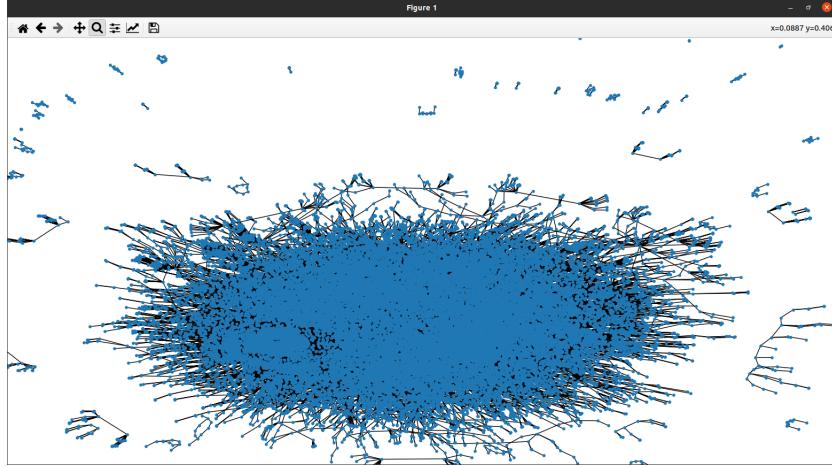
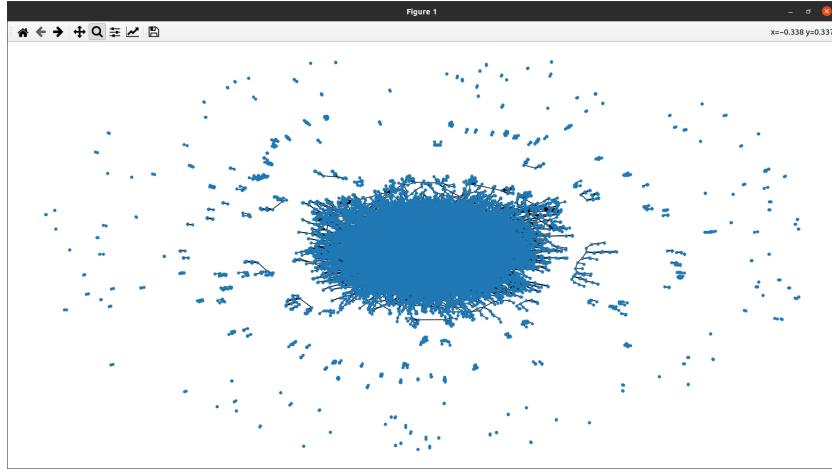


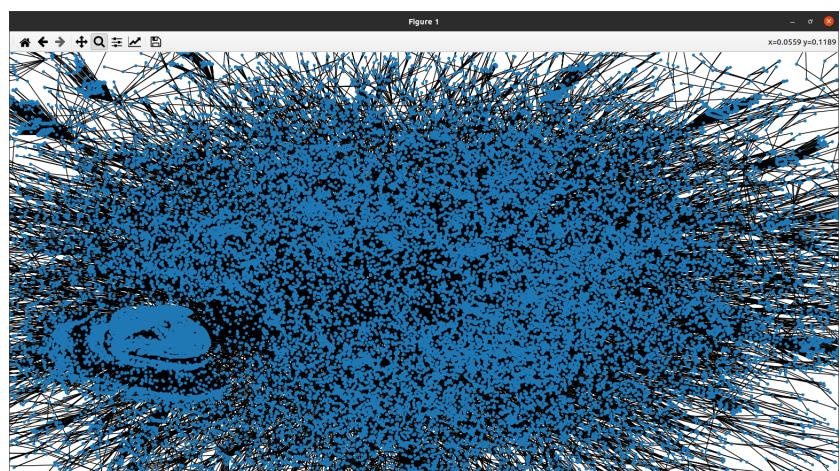
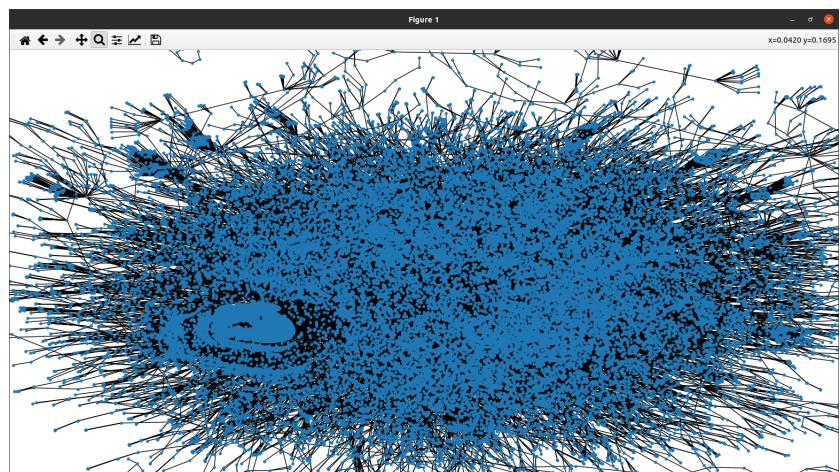
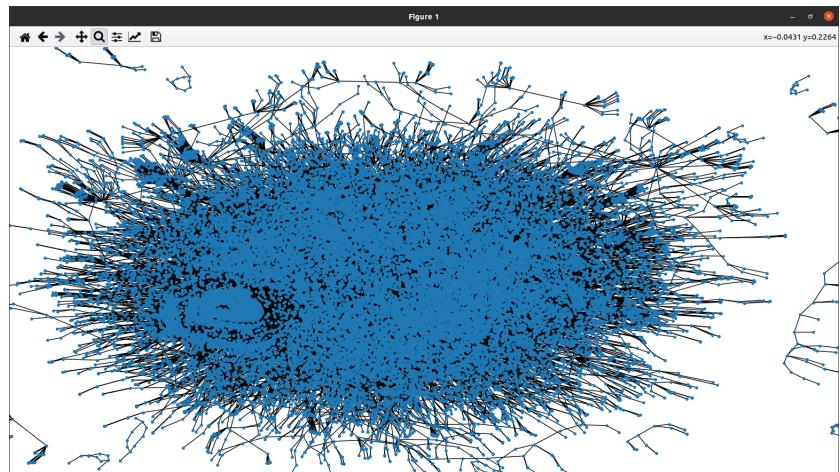
Figure 25: Vista documentos de edges y nodes.

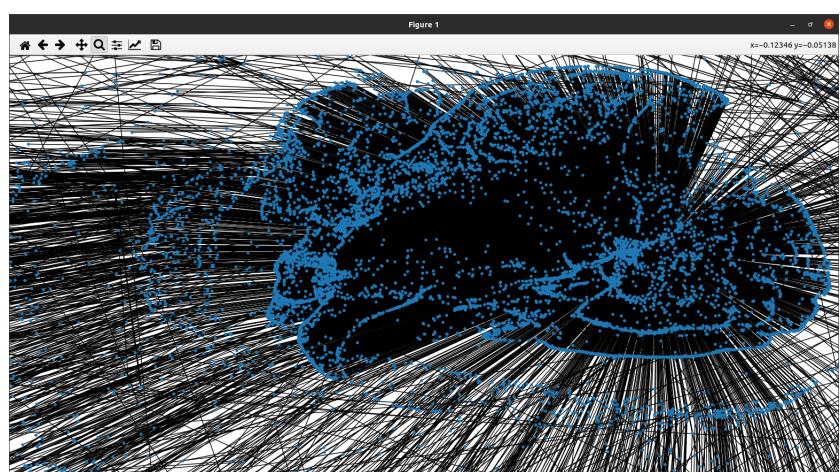
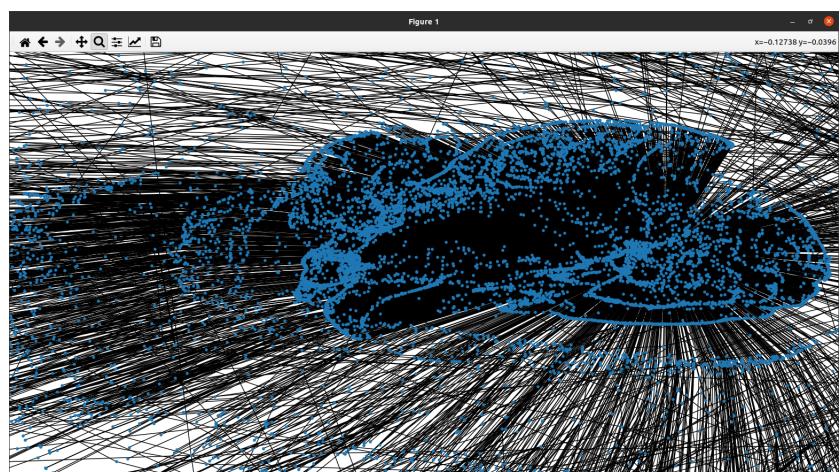
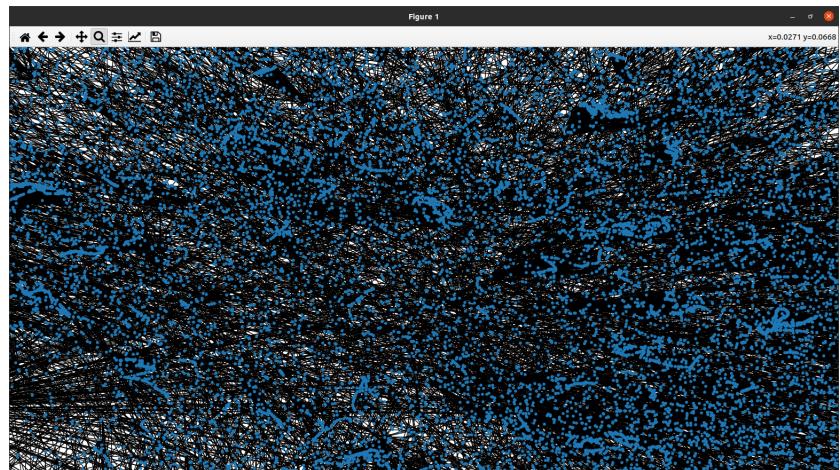
10 Pruebas de funcionamiento

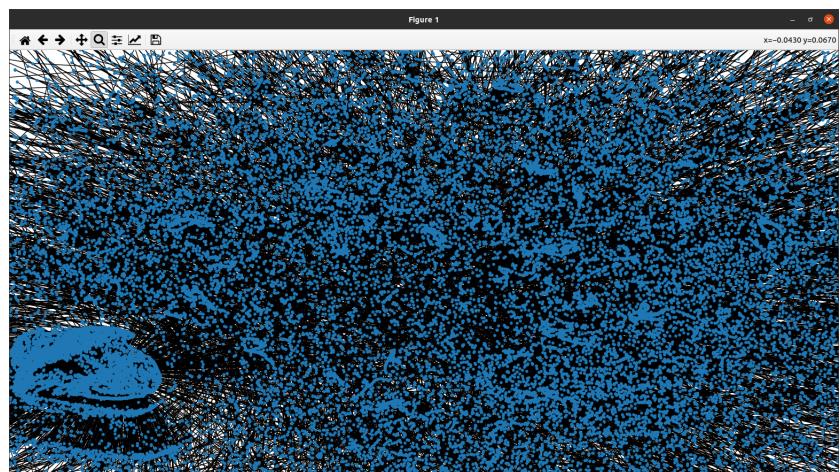
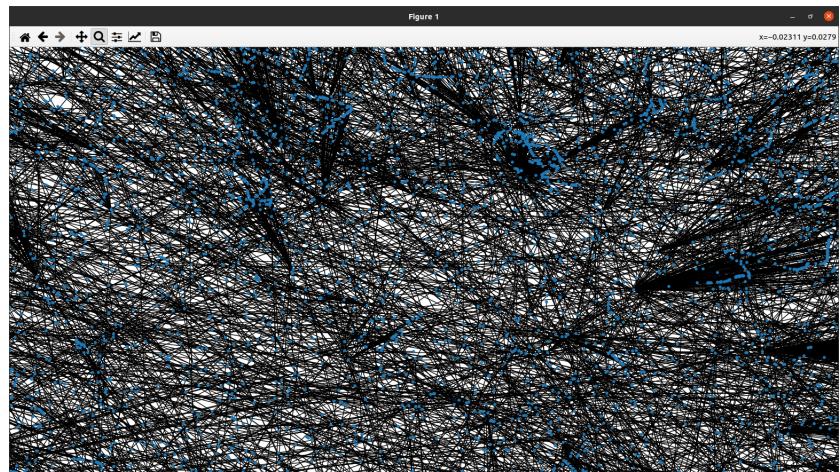
Para las pruebas de funcionamiento se utilizaron universo de $2x2$, $3x3$ y $4x4$ para la regla clásica de life $B3/S23$. Los resultados fueron los siguientes.

10.1 Universo de $4x4$ B3/S23

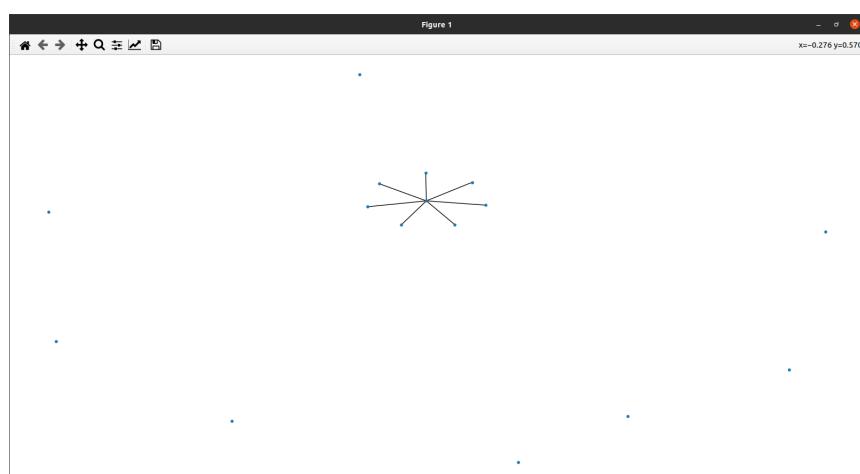




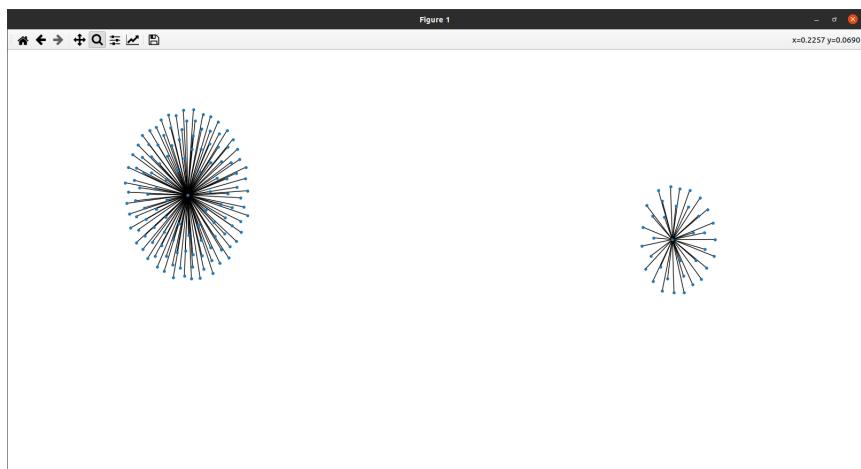
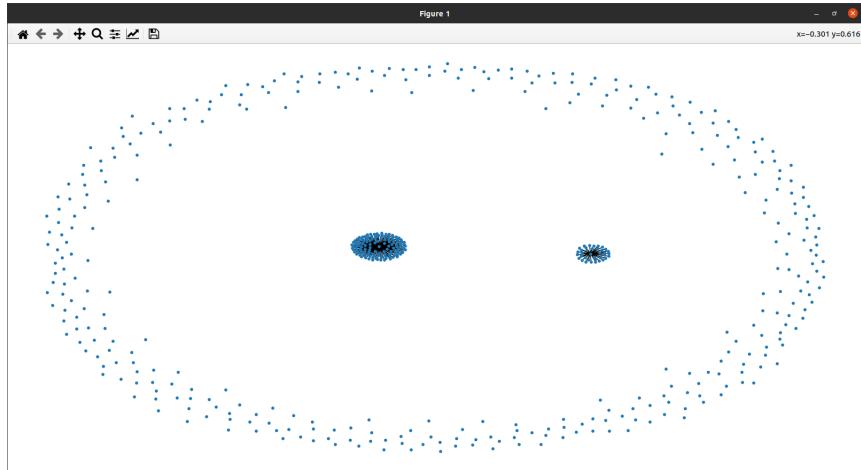


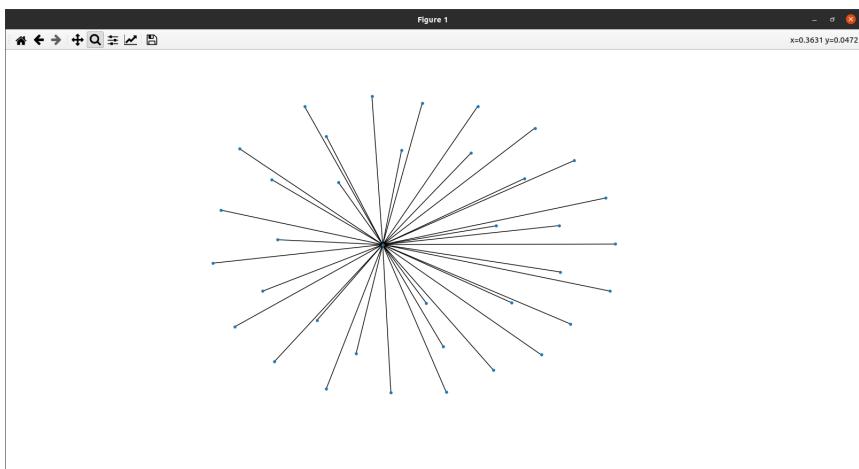
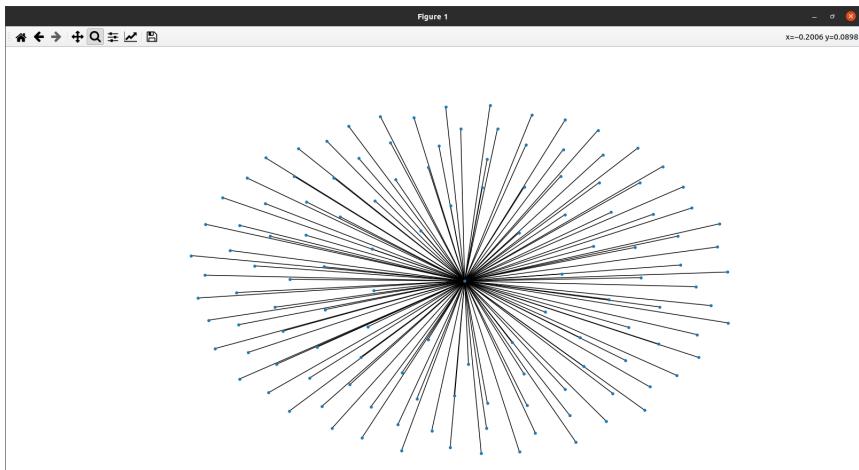


10.2 Universo de 2x2 B2/S3

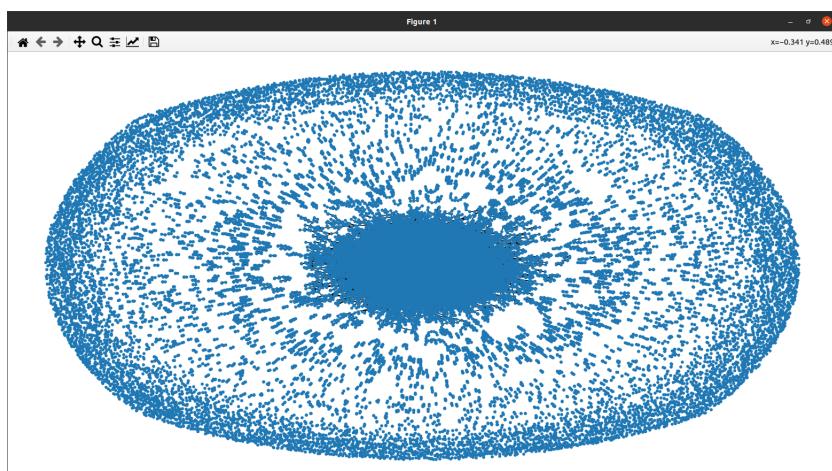


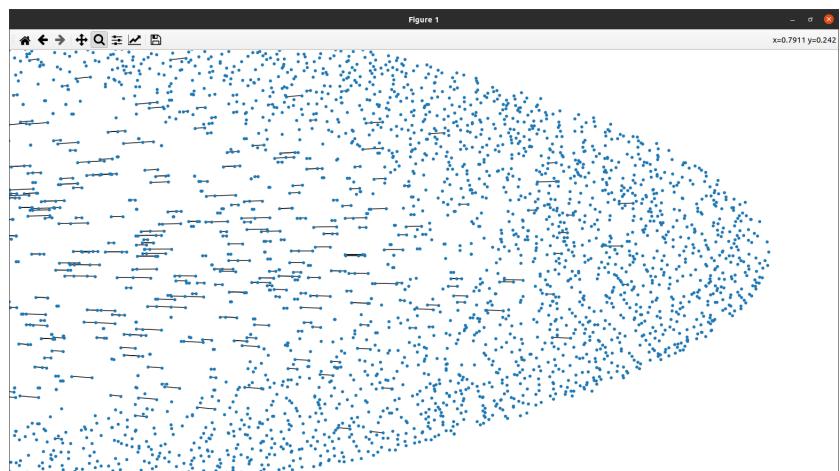
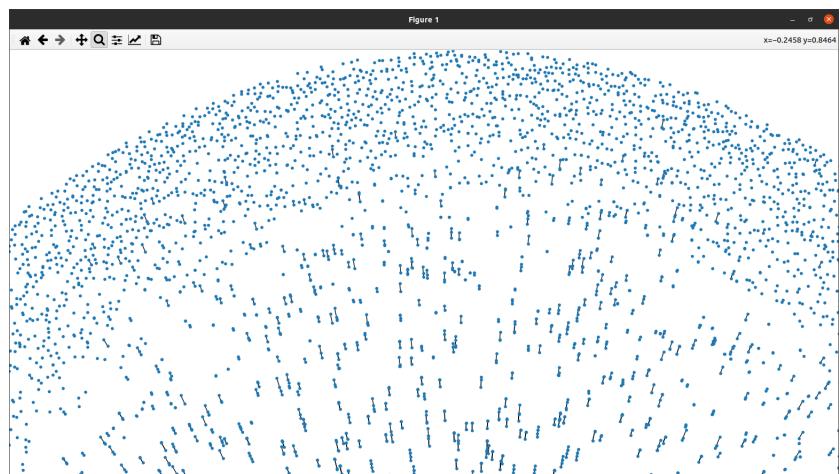
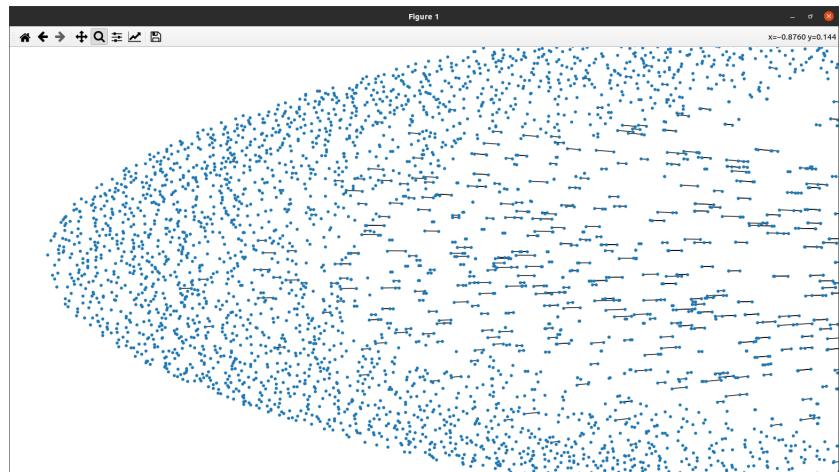
10.3 Universo de 3x3 B2/S3

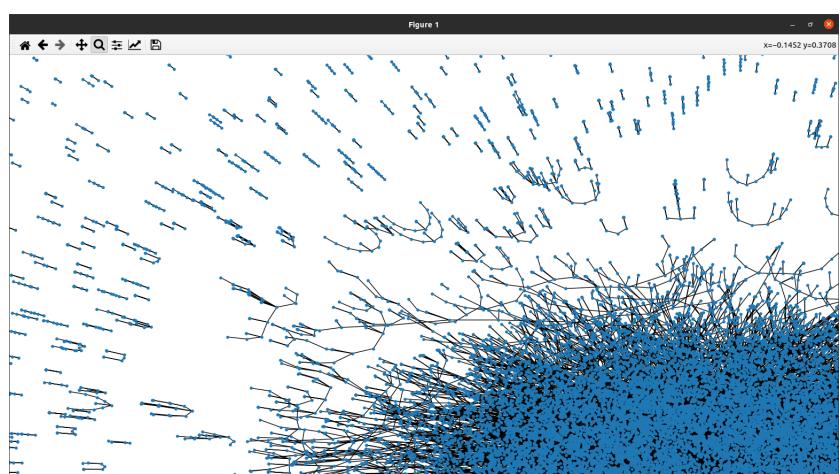
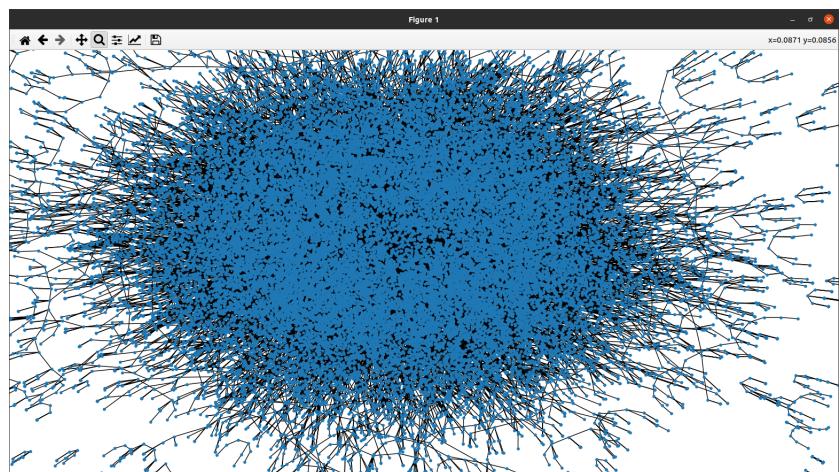
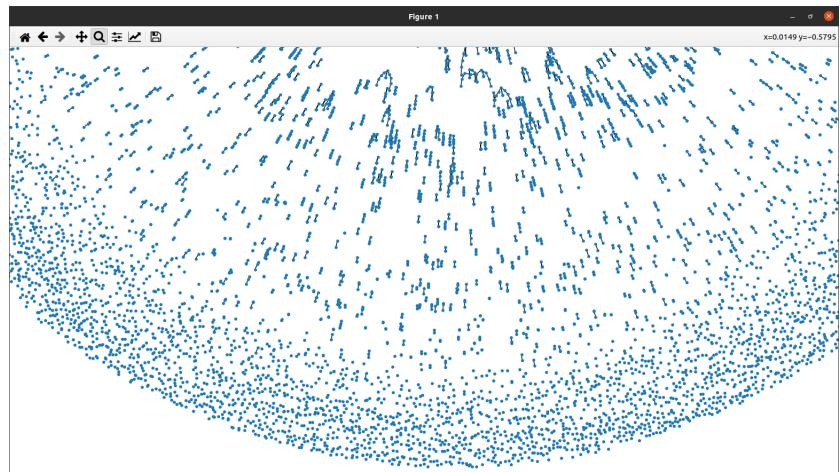


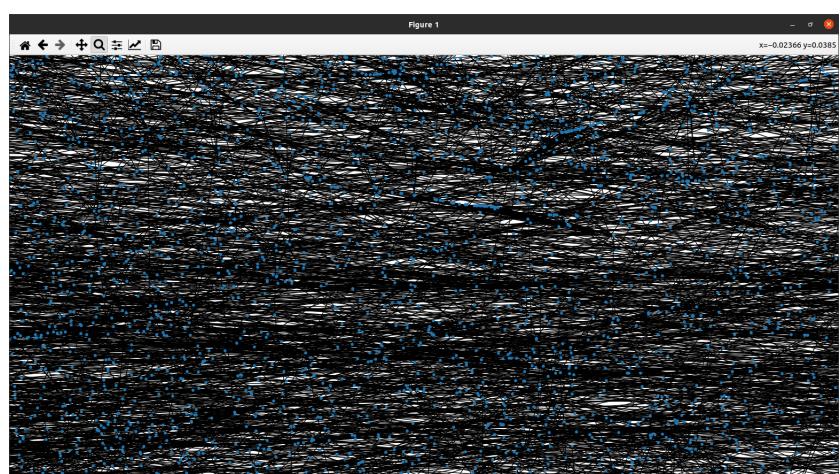
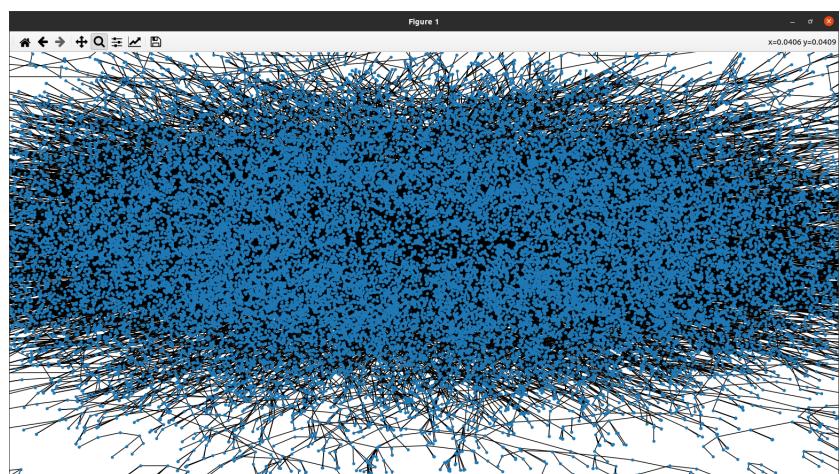
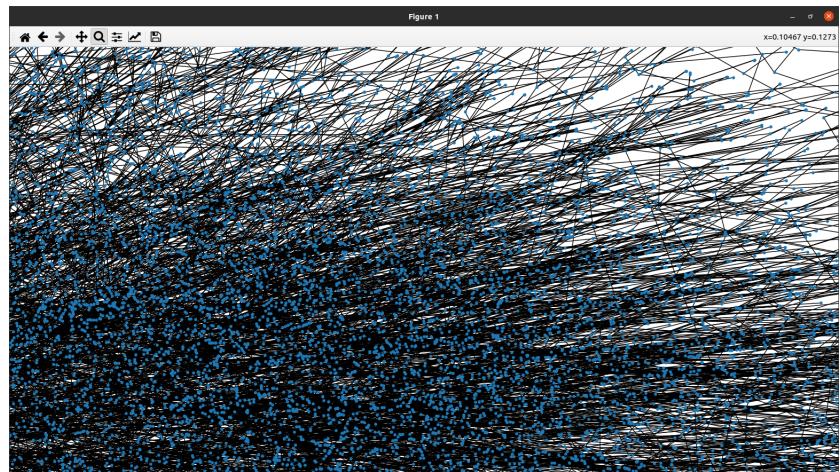


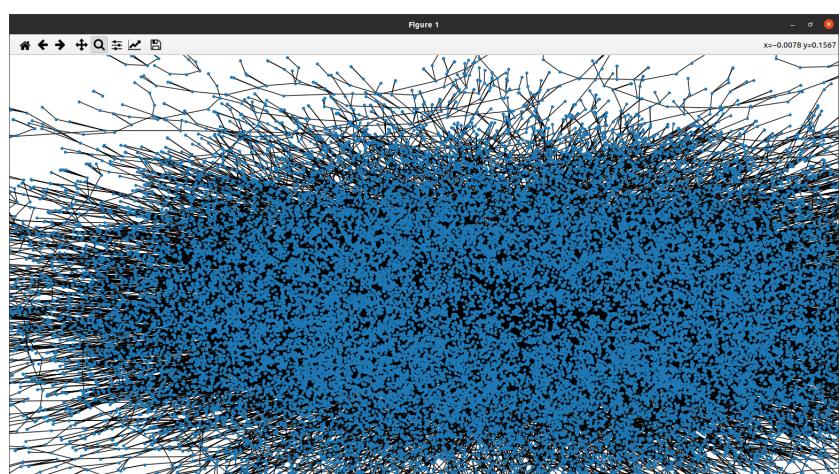
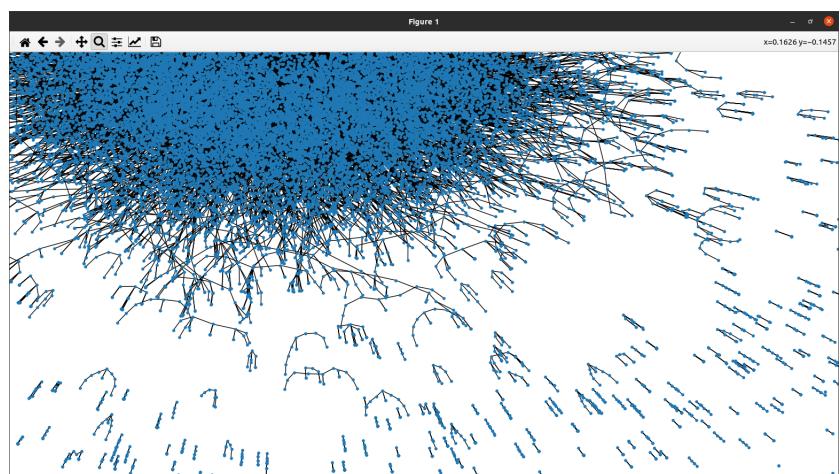
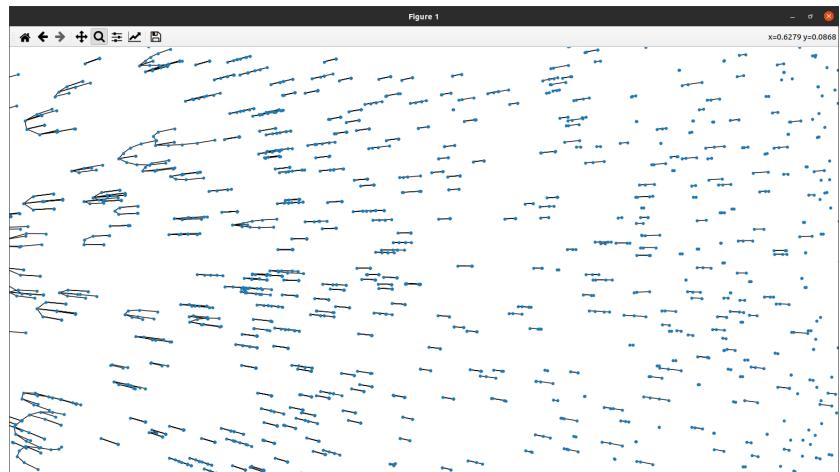
10.4 Universo de 4x4 B2/S3

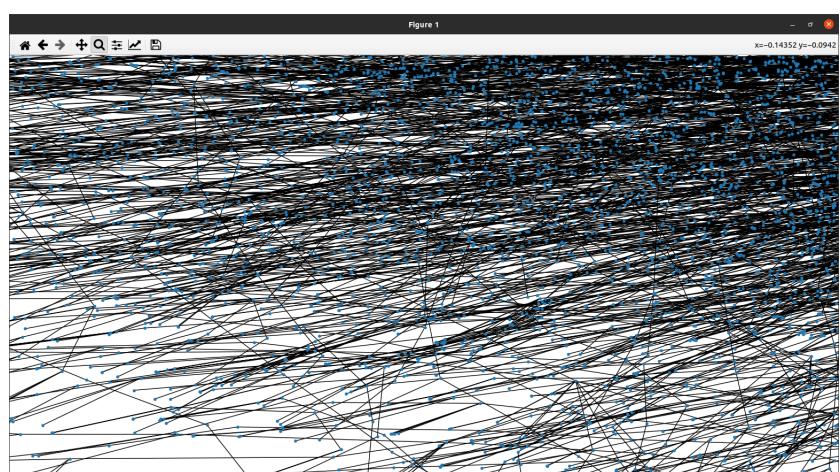
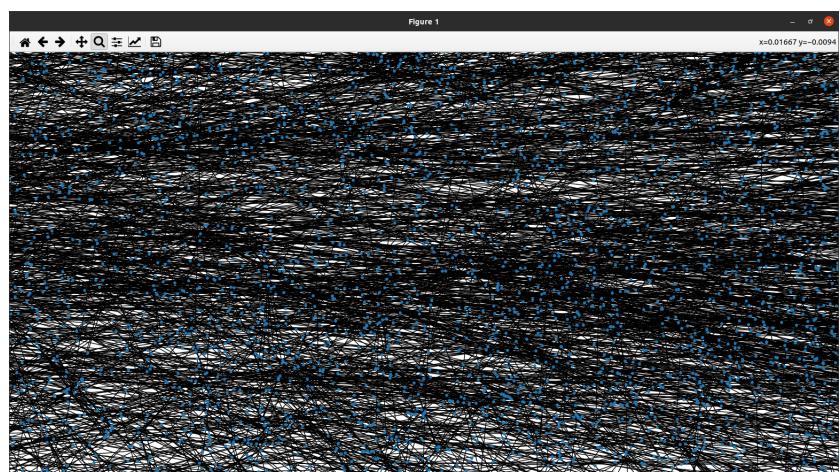


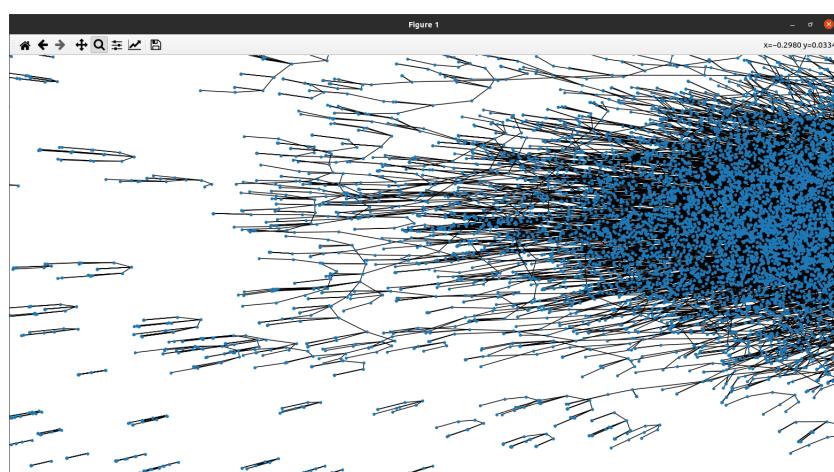
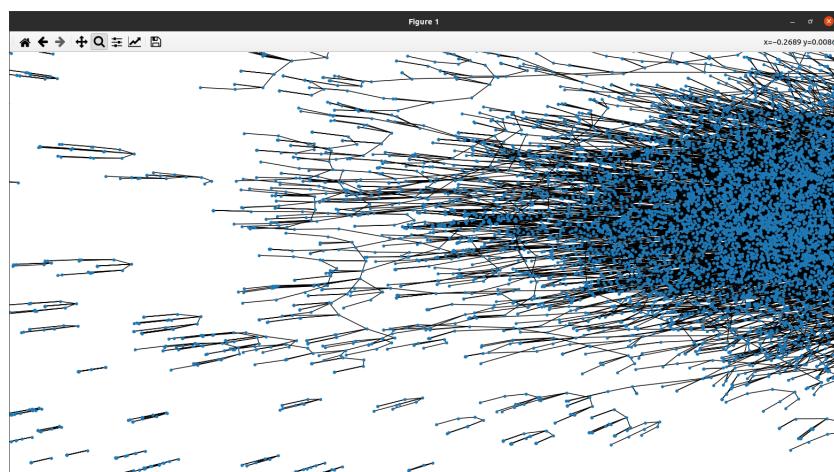
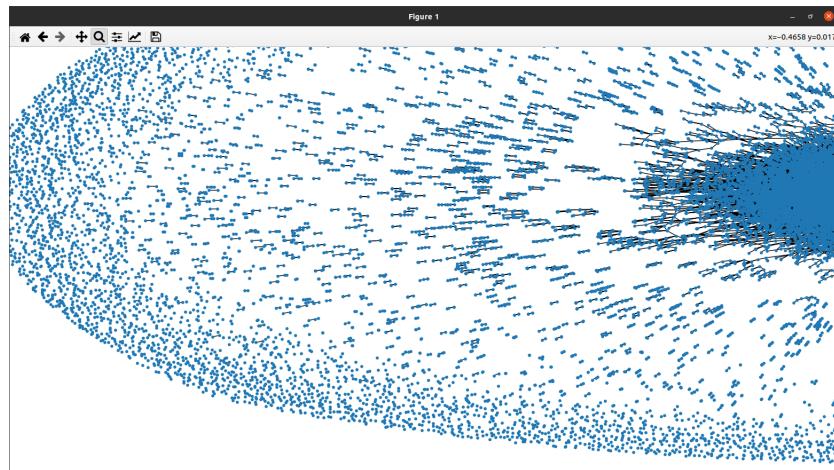


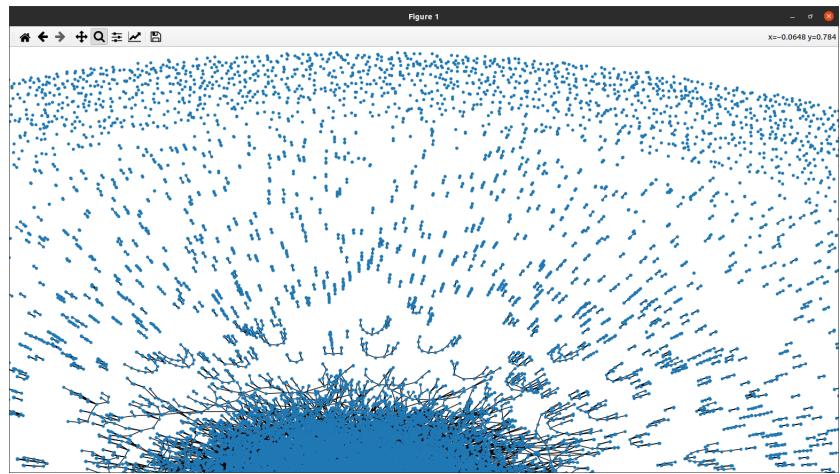




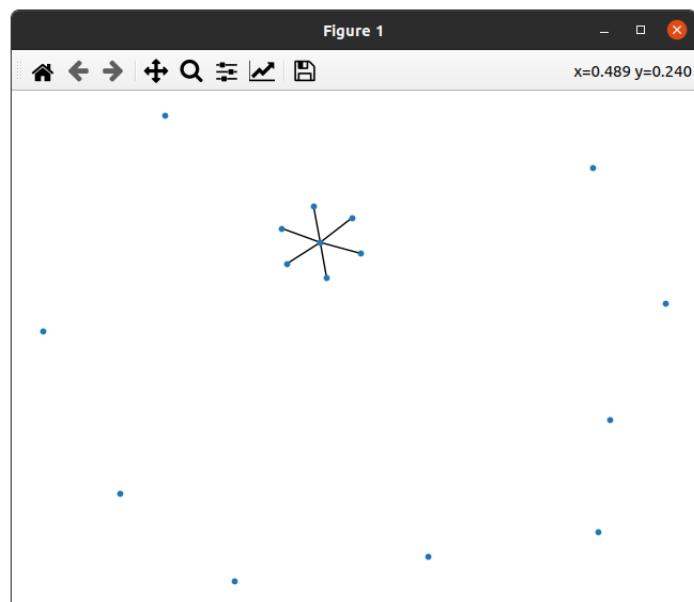




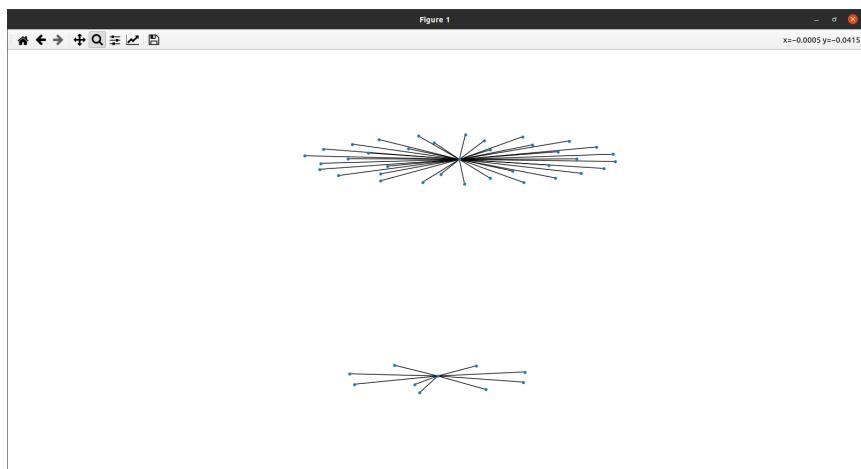
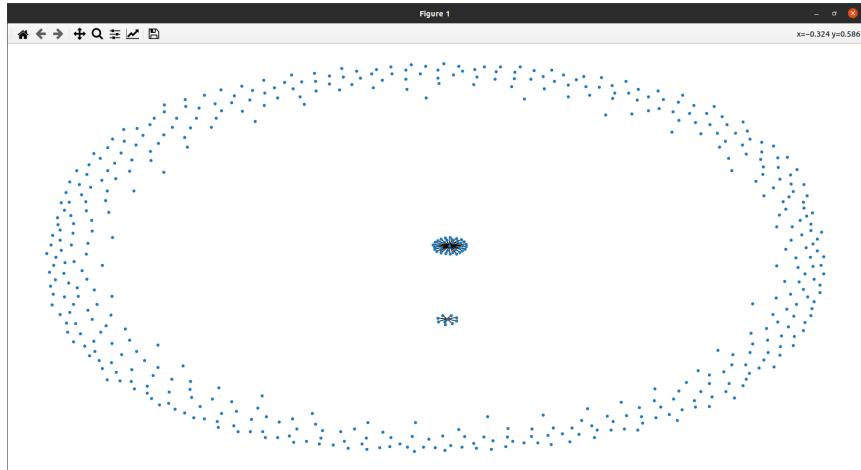




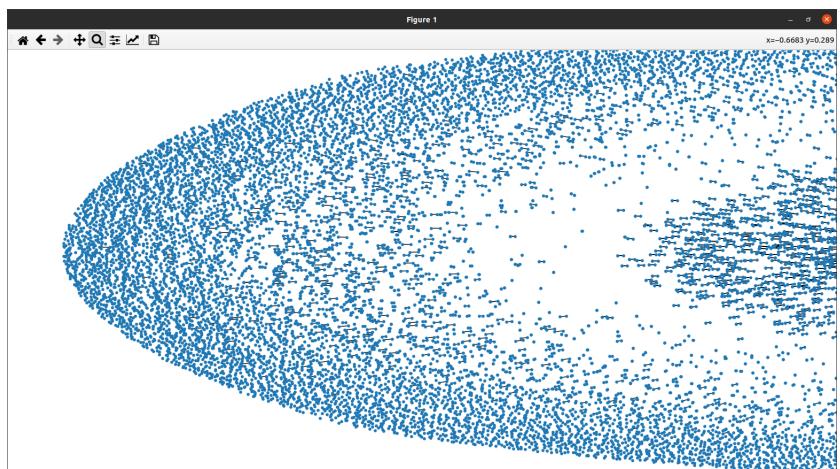
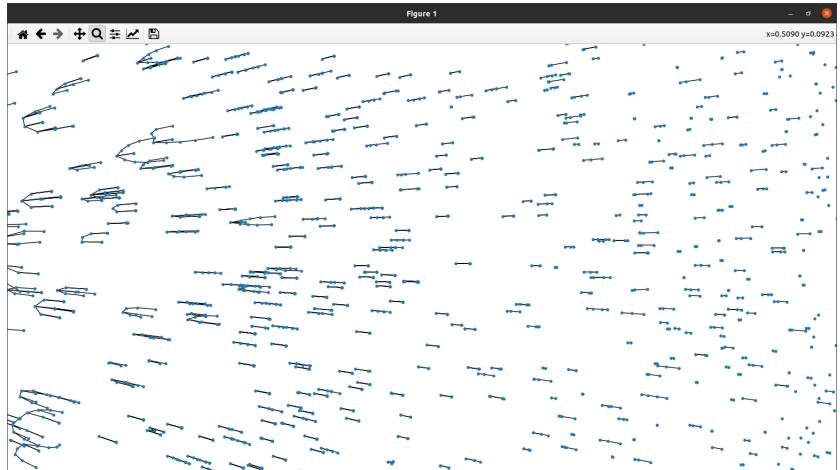
10.5 Universo de 2x2 B2/S7

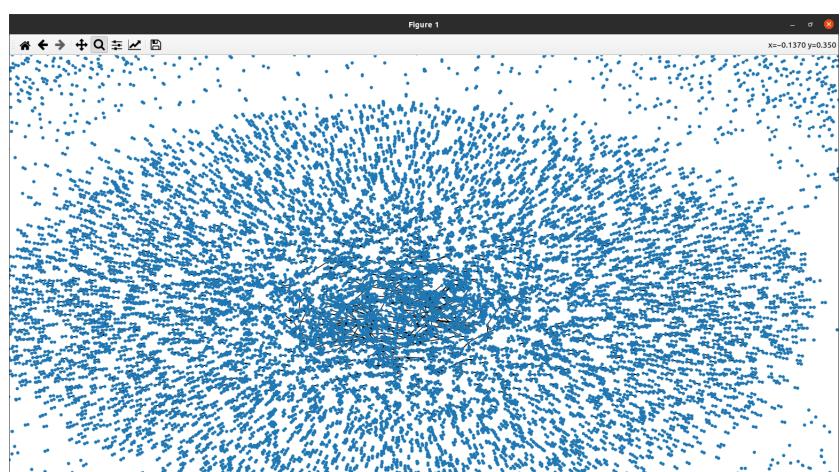
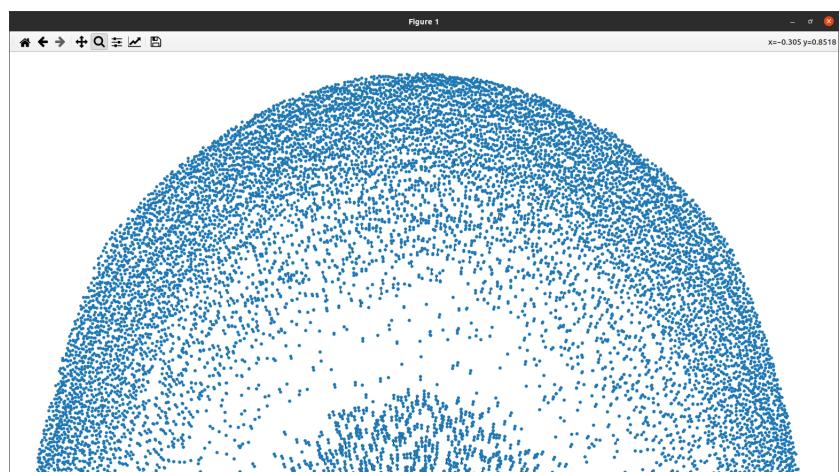
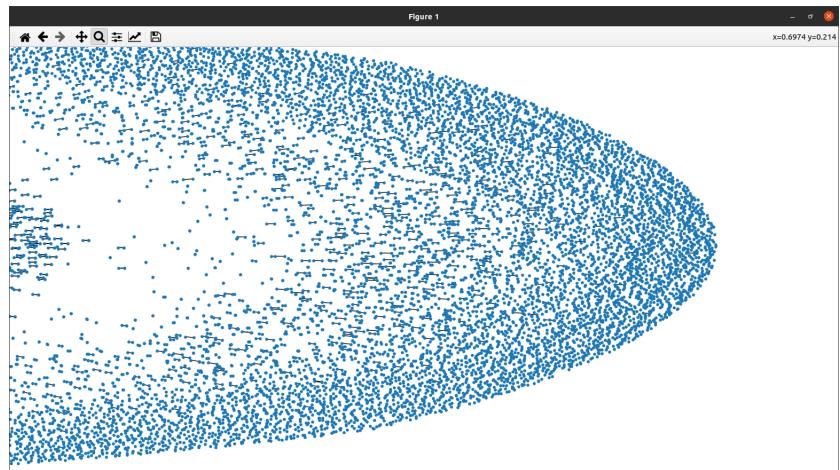


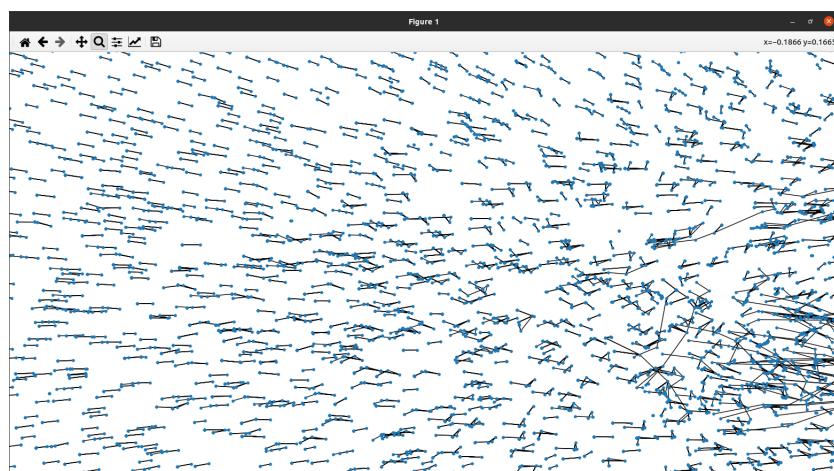
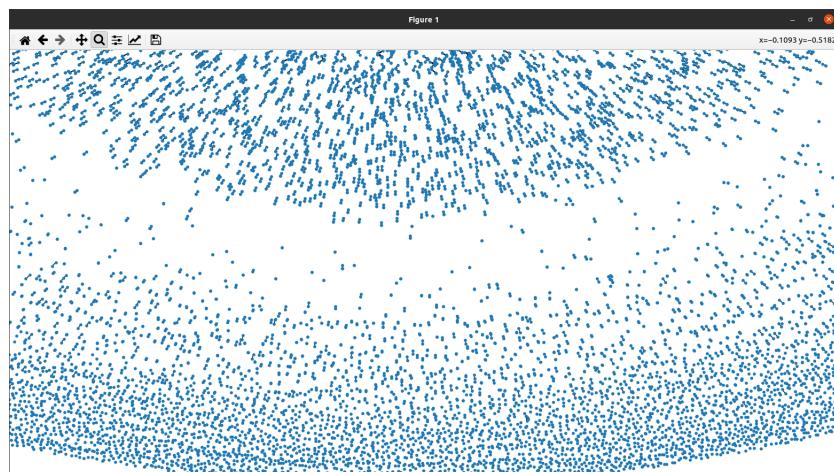
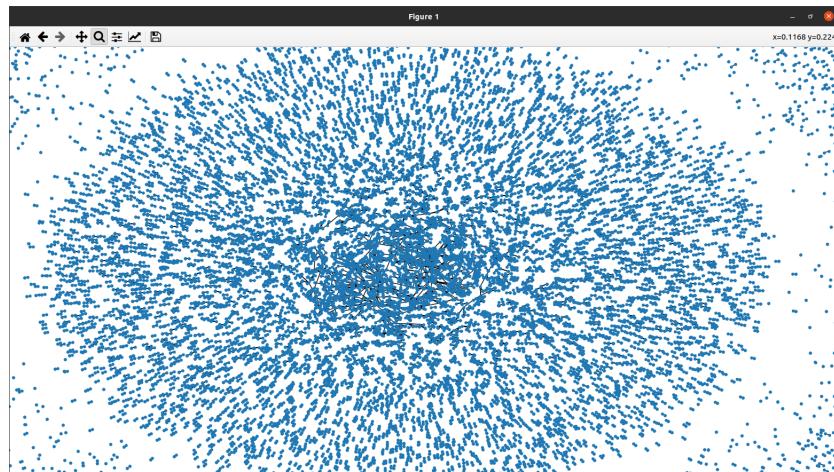
10.6 Universo de 3x3 B2/S7

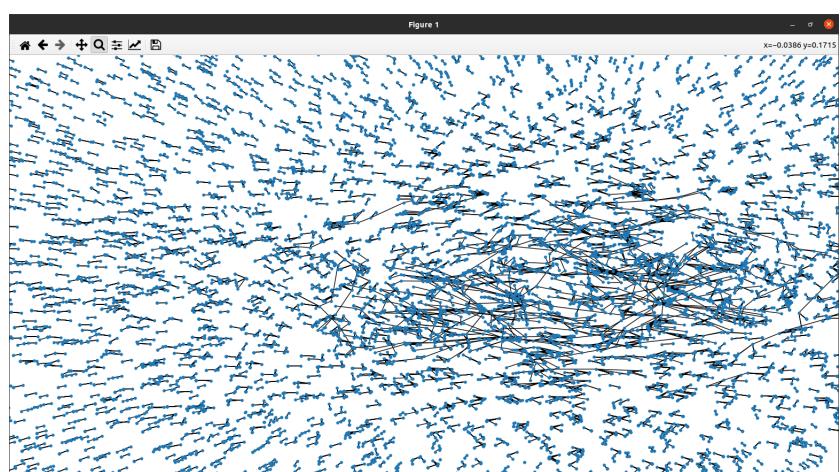
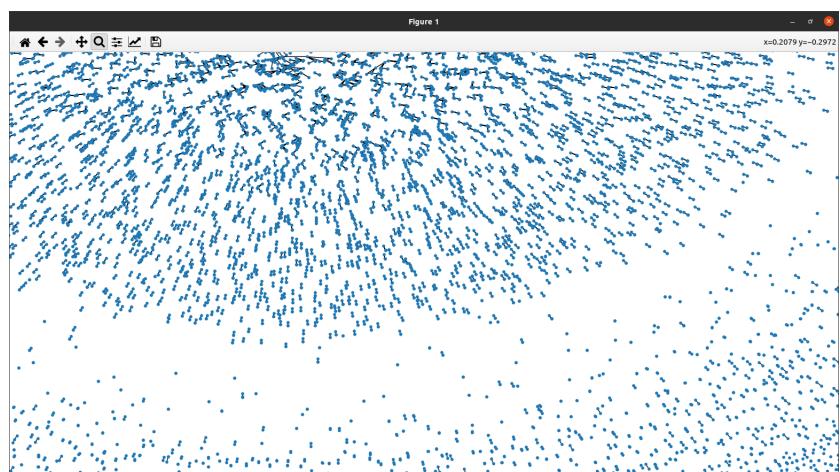
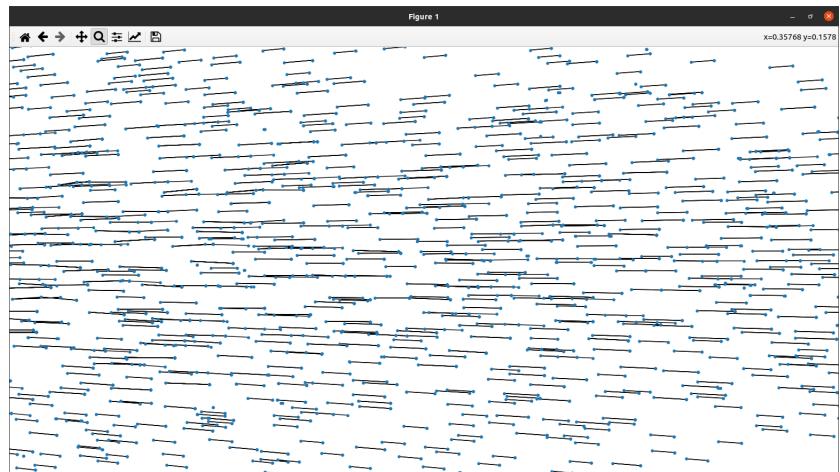


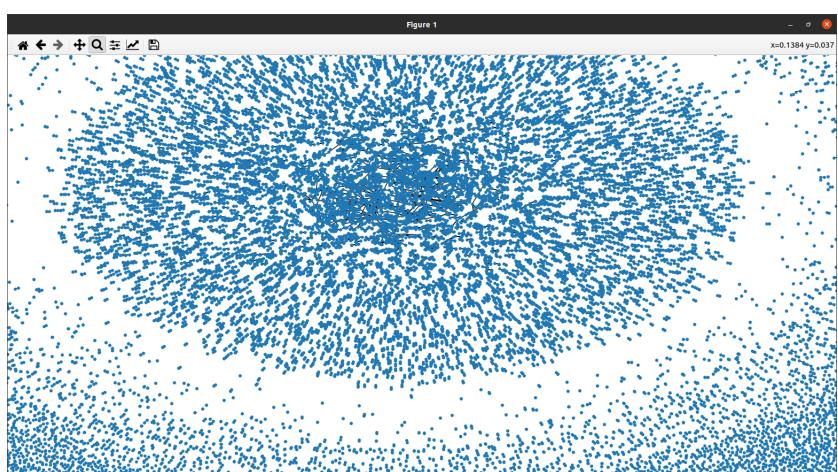
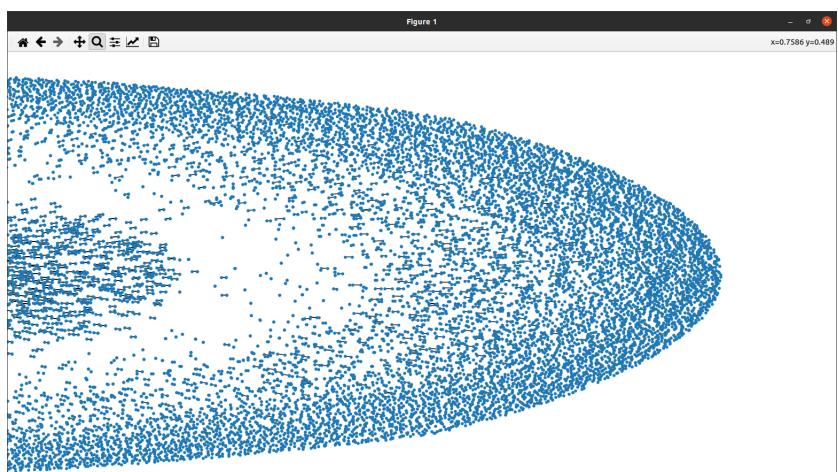
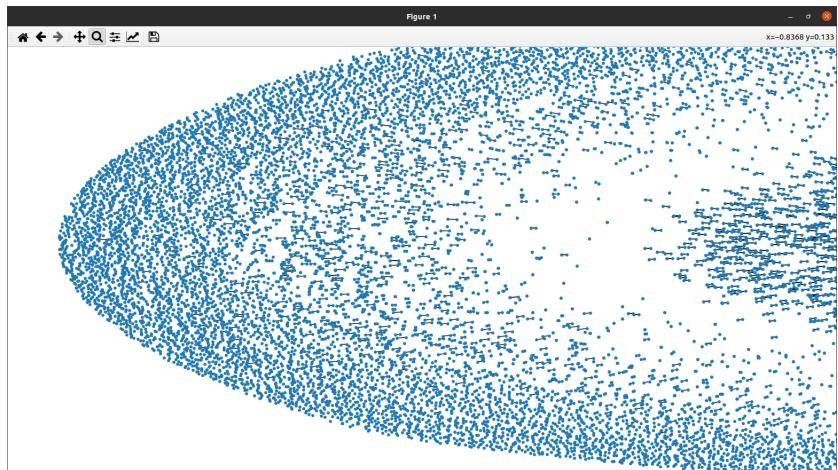
10.7 Universo de 4x4 B2/S7

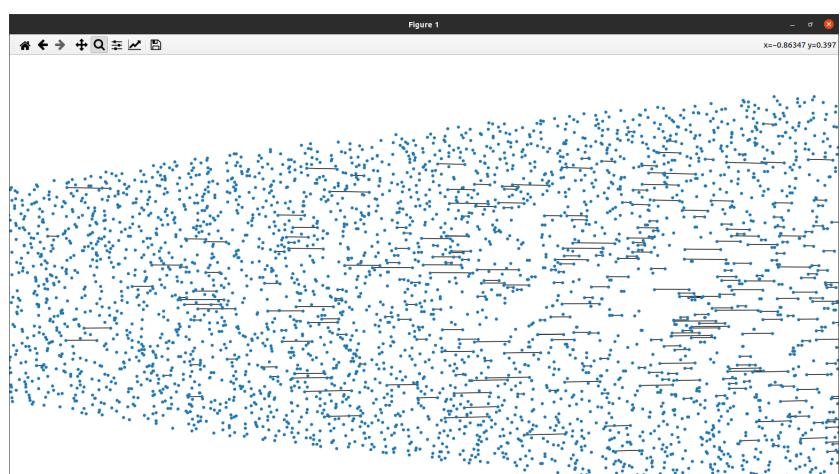
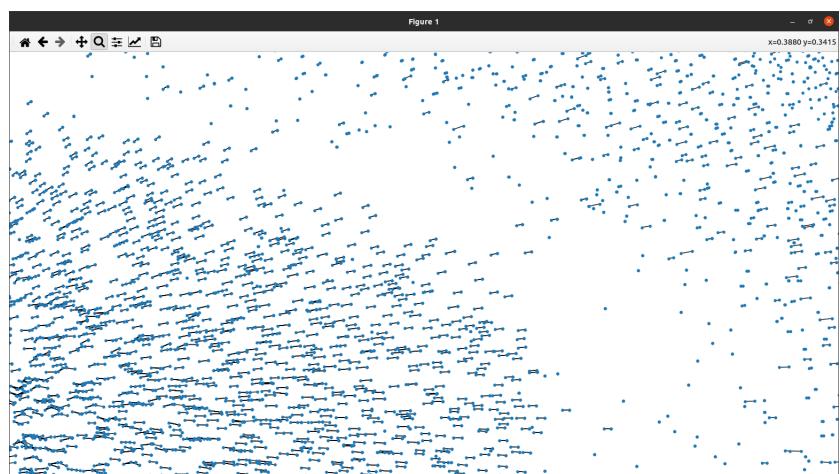
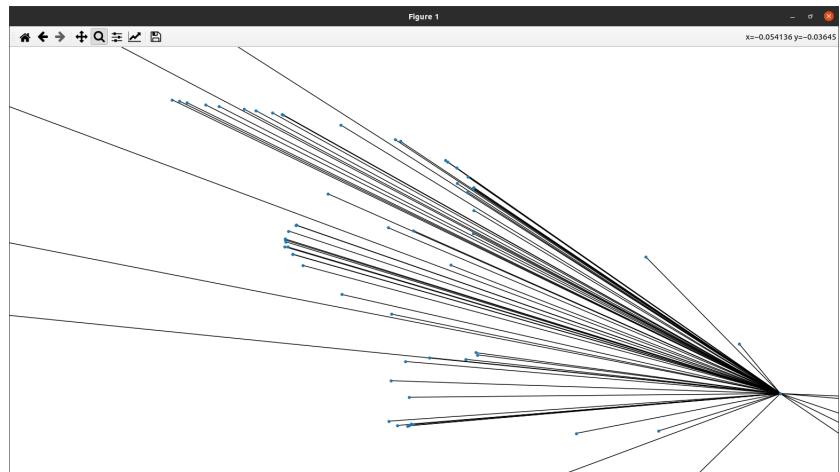


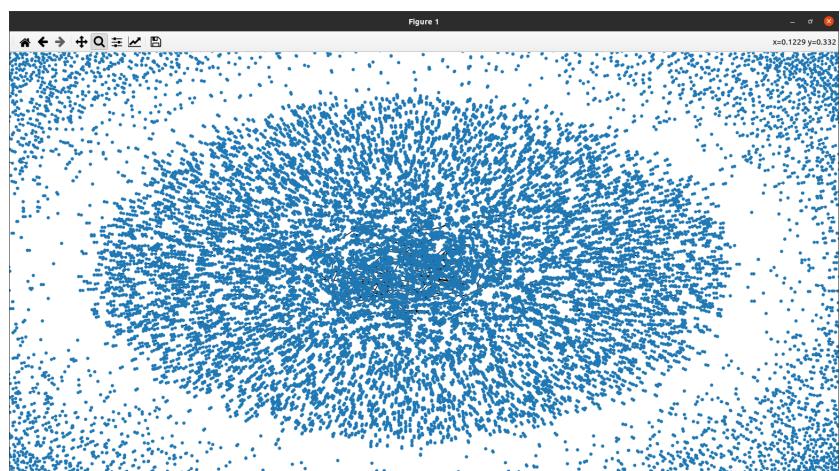
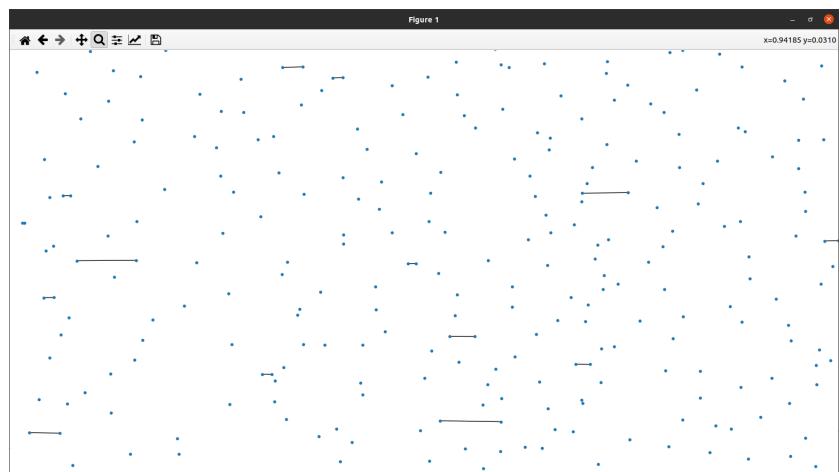
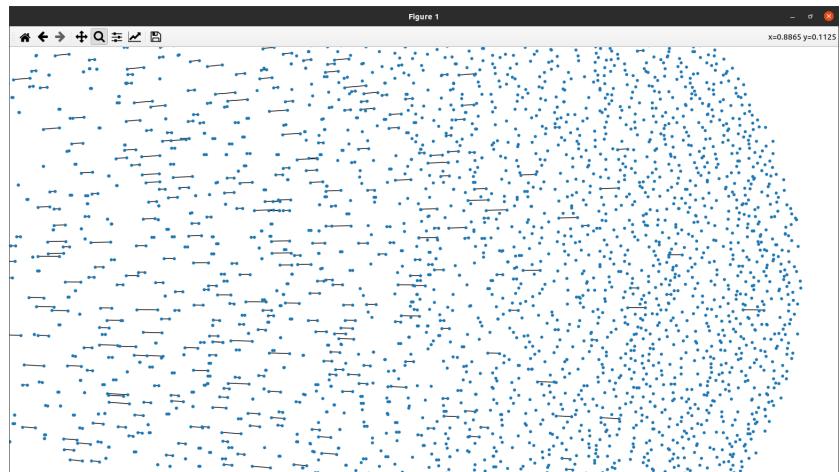


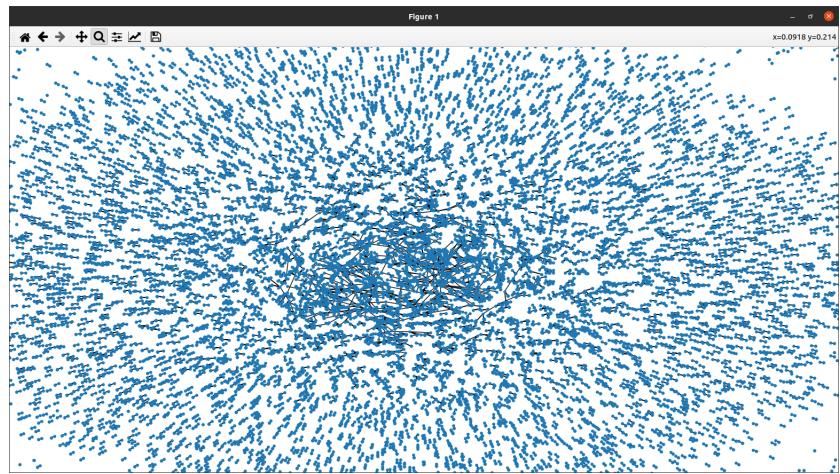












10.8 Código de implementación

```
1 #import pygame, sys
2 import numpy as np
3 import time
4 import math
5 #import pygame, sys
6 import networkx as nx
7 import matplotlib.pyplot as plt
8 import collections
9
10 #IMPORTAMOS TODA LA LIBRERIA
11
12
13 lenList = 0
14 combinationsList = []
15 globalCombination = ""
16 livingCells = 0
17 cases = []
18
19 min_nacimientos = 0
20 max_nacimientos = 0
21
22 min_survivencia = 0
23 max_survivencia = 0
24
25
26
27 BLACK = (0, 0, 0)
28 WHITE = (255,255,255)
29
30
31 def showList(combinationsList):
32     for x in range(0,len(combinationsList)):
33         print(combinationsList[x])
34
35
36 def getCombinations(lenList, combination):
37     global combinationsList;
38     newCombination = ""
39
40     for x in range(0,2):
41         if(lenList != 1):
42             newCombination = combination + str(x)
43             getCombinations(lenList-1, newCombination)
44             newCombination = ""
45         else:
46             newCombination = combination + str(x)
47             combinationsList.append(newCombination)
48             newCombination = ""
49     combination = ""
50
51
52 def stringToMatriz(combination):
53     column = []
54     row = []
55     global lenList
56     STOP = int(math.sqrt(lenList))
57     index = 0
58
59     for x in range(0,len(combination)):
60         for y in range(0,len(combination[x])):
```

```

61
62     if(index < STOP):
63         row.append(int(combination[x][y]))
64
65     if( index == STOP -1):
66         column.append(row.copy())
67         row.clear()
68         index = 0
69     else:
70         index += 1
71
72     cases.append(column.copy())
73     column.clear()
74
75
76
77 def copyListOfLists(combinationsList):
78     newCombinationsList = combinationsList.copy()
79
80     for c in range(0, len(combinationsList)):
81         newCombinationsList[c] = combinationsList[c].copy()
82
83     return newCombinationsList
84
85 def functionLife(lenList, combinationsList):
86
87     newCombinationsList = copyListOfLists(combinationsList)
88
89     for x in range(0,lenList ):
90         valuesX = []
91         valuesY = []
92         for y in range(0,lenList):
93             livingCells = 0
94
95             valuesX = [
96                 ((x-1) % lenList , (y-1) % lenList),
97                 ((x) % lenList, (y-1) % lenList),
98                 ((x+1) % lenList, (y-1) % lenList),
99                 ((x-1) % lenList, (y) % lenList),
100                ((x+1) % lenList, (y) % lenList),
101                ((x-1) % lenList, (y+1) % lenList),
102                ((x) % lenList, (y+1) % lenList),
103                ((x+1) % lenList,(y+1) % lenList)
104
105            ]
106            valuesX = set(valuesX)
107
108            for index in valuesX:
109                a,b = index
110                livingCells = livingCells + int(combinationsList[a][b])
111
112            #BORN
113            if ((combinationsList[x][y]) == 0 and ((livingCells == min_nacimientos)):
114                newCombinationsList[x][y] = 1
115            #SURVIVE
116            elif((combinationsList[x][y] == 1) and ((livingCells < min_supervivencia
117 ) or (livingCells > max_supervivencia))):
118                newCombinationsList[x][y] = 0
119
120    return newCombinationsList
121
122 def sumElementsList(list):

```

```

121     total = 0
122     for x in list:
123         total = total + sum(x)
124     return total
125
126 def createGraph():
127
128     global min_nacimientos
129     global max_nacimientos
130     global min_survivencia
131     global max_survivencia
132     global lenList
133
134     lenList = (int)(v.get())*(int)(v.get())
135
136     separado = (nacimiento1.get()).split("/")
137     min_nacimientos = (int)(separado[0][1])
138     max_nacimientos = (int)(separado[0][1])
139
140     if(len(separado[1]) == 2):
141         min_survivencia =(int)(separado[1][1])
142         max_survivencia =(int)(separado[1][1])
143     else:
144         min_survivencia =(int)(separado[1][1])
145         max_survivencia =(int)(separado[1][2])
146
147
148     ventan.destroy()
149
150
151     getCombinations(lenList, globalCombination )
152     print("Generated combination ",len(combinationsList))
153
154     stringToMatriz(combinationsList)
155
156     graph = []
157
158     G = nx.Graph()
159
160
161     decimal = 0
162     path = "nodes.txt"
163     n = open(path, 'w')
164
165     path = "edges.txt"
166     e = open(path, 'w')
167
168     ciclos = 0
169     ciclo = []
170
171     for x in range(0, len(cases)):
172         decimal = cases.index(cases[x])
173         variableFunctionLife = cases[x].copy()
174         ciclo.append(decimal)
175         iteracion = 0
176         antecesor = 0
177         sem = 0
178         while(True):
179
180             if(decimal in G.nodes):
181
182                 if(iteracion == 0):

```

```

183     G.add_node(decimal)
184     n.write(str(decimal))
185     antecesor=decimal
186     variableFunctionLife = functionLife(len(variableFunctionLife),
187     variableFunctionLife)
188     decimal = cases.index(variableFunctionLife)
189
190     ciclo.append(decimal)
191
192     iteracion+=1
193
194     if(ciclo[0] == ciclo[len(ciclo)-1]):
195         print(ciclo)
196         ciclos +=1
197         ciclo.clear()
198         #print(" equal ")
199         G.add_edge(decimal, antecesor)
200         union = str(decimal) + " "+str(antecesor) + "\n"
201         #print(union)
202         e.write(union)
203
204         break
205     else:
206         #introducimos nodo a grafo
207         iteracion+=1
208         G.add_node(decimal)
209
210         n.write(str(decimal)+ "\n")
211
212         #Verificamos si existe antecesor
213         if(iteracion != 1):
214             union = str(decimal) + " " + str(antecesor) + "\n"
215             e.write(union)
216             G.add_edge(decimal,antecesor)
217
218         #aplicamos life
219         antecesor=decimal
220         variableFunctionLife = functionLife(len(variableFunctionLife),
221         variableFunctionLife)
222         decimal = cases.index(variableFunctionLife)
223
224         ciclo.append(decimal)
225
226         n.close()
227         e.close()
228
229         print("Ciclos totales: ", ciclos)
230         print("-----")
231
232         nx.draw(G, node_size= 10)
233         plt.show()

```

11 Conclusiones finales

El desarrollo de este proyecto me ha parecido de bastante valor, principalmente porque durante el desarrollo, fui aprendiendo bastantes cosas que no sabia un ejemplo un claro ejemplos es el uso de la librería Networkx para graficar grafos que al principio me costo un poco de trabajo pero después de algunas horas de

hacer algunas pruebas logre utilizar la librería para graficar mis grafos. También aprendí el uso de la biblioteca pygame otra librería que me ayudo mucho en el desarrollo del proyecto, además esta librería me dejó bastante interesado para seguir programando con ella.

También termine aprendiendo lo importante que es optimizar el código porque a lo largo del desarrollo me tope con problemas que hacían que mi computadora dejara de trabajar, por ejemplo en la ultima parte del proyecto cuando tenia que generar un universo de $5 * 5$ mi computadora genero las 33554432 combinaciones posibles dentro del universo pero no pudo procesarlas porque después de un tiempo se congelaba mi maquina, intente solucionarlo pero no pude lograr que mi computadora no se congelara, así que tuve que quedarme hasta el universo $4 * 4$.

Creo que la parte que me dio menos problemas fue la simulación, ya que aquí si logre trabajar con universos mayores a los $1000 * 1000$ sin problemas.

En lo que respecta a la parte teórica del proyecto, termine entendiéndola mejor porque en las clases con el profesor resultaba un poco complicado abstraer algunos conceptos lo que terminaba confundiéndome, pero conforme avanzaba en el proyecto mis dudas fueron desapareciendo, principalmente porque fui viendo como se hizo el análisis para llegar a ciertos conceptos. Por ejemplo la parte de generar los atractores se me hizo interesante porque al tener todas la combinaciones de un universo estamos descubriendo cual es el comportamiento completo que tiene un sistema con ciertas reglas y a donde va cuando aparece una combinación en específico. Por otra parte al final del proyecto me di cuenta del poder y la importancia que tienen los atractores no solo en la computación si no en otras áreas como la Biología.

12 Referencias

1. colaboradores de Wikipedia. (2020, 14 octubre). Juego de la vida. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Juego_de_la_vida
2. Networkx - Networkx Documentation. (2018, 1 octubre). Rip Tutorial. <https://networkx.org/>
3. Matplotlib - Matplotlib | Matplotlib: Visualization with Python. (2012, 10 febrero). Matplotlib. <https://matplotlib.org/>