
INSTITUTO POLITÉCNICO NACIONAL,
ESCUELA SUPERIOR DE CÓMPUTO

Autómata Celulares

SISTEMAS COMPLEJOS

Oscar Andrés Rosas Hernandez

9 de mayo de 2020

Índice general

I Marco Teórico	2
1. Autómata celulares	3
1.1. Definición	4
1.1.1. Características	4
2. Diagramas de Atracción	5
2.1. Crear una enumeración	5
2.2. Creando el simulador de Game of Life like en Rust	6
3. El Juego de la vida: R(2, 3, 3, 3)	9
3.1. 2x2	9
3.2. 3x3	11
3.3. 4x4	13
3.4. 5x5	15
4. Difusión: R(7, 7, 2, 2)	16
4.1. 2x2	16
4.2. 3x3	18
4.3. 4x4	19
4.4. 5x5	20

Parte I

Marco Teórico

Capítulo 1

Autómata celulares

1.1. Definición

Un autómata celular es un sistema dinámico discreto que consiste en una red regular de autómatas (celdas) de estado finito que cambian sus estados dependiendo de los estados de sus vecinos (y del mismo), de acuerdo con una función de transferencia.

Todas las células cambian su estado simultáneamente usando la misma regla de actualización. El proceso se repite en pasos de tiempo discretos. Resulta que con reglas de actualización sorprendentemente simples se pueden producir dinámicas extremadamente complejas como en el famoso Juego de la vida de John Conway. [1]

El aspecto que mas los caracteriza es su capacidad de lograr una serie de propiedades que surgen de la propia dinámica local a través del paso del tiempo y no desde un inicio, aplicándose a todo el sistema en general. Por lo tanto, no es fácil analizar las propiedades globales de un AC desde su comienzo, complejo por naturaleza, si no es por medio de una simulación, partiendo de un estado o configuración inicial de células y cambiando en cada instante los estados de todas ellas de forma síncrona.

1.1.1. Características

- Son discretos tanto en tiempo como en espacio
- Son homogeneos tanto en tiempo como en espacio (la misma regla es aplicada a todas las celulas al mismo tiempo)
- Sus interacciones son locales

Para especificar una autómata celular, debemos especificar los siguientes elementos (algunos de los cuales pueden ser claros por el contexto):

- La dimensión $d \in \mathbb{Z}^+$,
- El conjunto de estados finitos S
- Una vecindad N de celdas
- La función de activación $f : S^m \rightarrow S$

Por lo tanto, definimos formalmente a un autómata celular correspondiente como la 4-tupla $A = (d, S, N, f)$.

Capítulo 2

Diagramas de Atracción

Los diagramas de atracción son un sistema que nos permiten comprender de mejor manera como es que se relacionan los posibles estados de un automata.

Estos son especialmente buenos para encontrar lo que se conoce como puntos estacionarios en los que un estado tras uno o una serie de pasos cae en un ciclo finito de estados por el que pasa indefinidamente.

Ademas como su nombre lo indica nos permiten ver de manera sencilla los atractores, es decir estamos que estamos conectados a muchos mas.

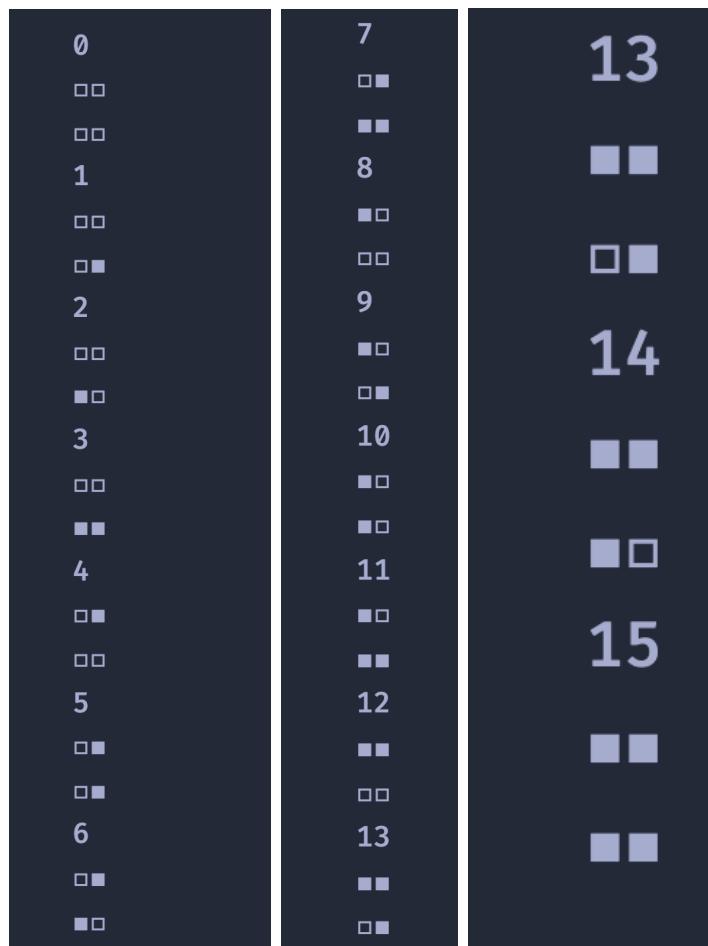
Para crearlos seguimos varios pasos.

2.1. Crear una enumeración

Lo primero que vamos a hacer es tener una forma de enumerar los posibles estados en los que puede estar el universo de nuestro automata, es muy importante ser claro en esta enumeración, pues una vez creada solo vamos a trabajar con enteros.

Para hacerla tome como inspiracion la tecnica de spagetification, esta consiste en tomar nuestro grid en 2d y apastarlo para conseguir un grid en 1D, luego como es que los automatas que vamos a ver solo tienen una sola dimension entonces podemos hacer algo muy sencillo, considerar ese tira 1D de estados y entenderla como un secuencia de bits, con ella podemos llegar a un numero, veamos un ejemplo con un 2x2.

Ahora bien, con la enumeración lista podemos movernos a la parte principal.



2.2. Creando el simulador de Game of Life like en Rust

Ahora vamos a dar un repaso rapido a como creamos nuestro simulador en Rust:

```
#[derive(Clone, Copy)]
pub enum Cell {
    Dead = 0,
    Alive = 1,
}

#[derive(Clone)]
pub struct Universe {
    width: usize,
    height: usize,
    cells: Vec<Cell>,
    survive_min: u8,
    survive_max: u8,
    birth_min: u8,
    birth_max: u8,
}
```

En esta parte del codigo vamos a definir lo que es una celda y el universo, que contiene un grupo de celdas asi como los parametros de la regla: $R(s_{min}, s_{max}, b_{min}, b_{max})$.

Recordemos que esta notacion quiere decir que una celula viva va a seguir viva si solo si

esta entre s_{min}, s_{max} y que una celula muerta va a morir si y solo si esta entre b_{min}, b_{max} .

De esta manera el simulador es bastante generico.

Ahora con esto listo estamos listo para la siguiente parte:

```
impl Universe {
    fn get(&self, row: usize, col: usize) -> Cell {
        let index = (row * self.width + col) as usize;
        self.cells[index]
    }

    fn count_alive_around(&self, row: usize, col: usize) -> u8 {
        let mut count = 0;
        for delta_row in &[0, 1, self.height - 1] {
            for delta_col in &[0, 1, self.width - 1] {
                let neighbor_row = (row + delta_row) % self.height;
                let neighbor_col = (col + delta_col) % self.width;
                count += self.get(neighbor_row, neighbor_col) as u8;
            }
        }
        count - self.get(row, col) as u8
    }

    pub fn tick(&mut self) {
        let mut next = Vec::with_capacity(self.width * self.height);

        for row in 0..self.height {
            for col in 0..self.width {
                let cell = self.get(row, col);
                let live_neighbors = self.count_alive_around(row, col);

                let next_cell = match (cell, live_neighbors) {
                    (Cell::Alive, neighbors) if neighbors < self.survive_min => Cell::Dead,
                    (Cell::Alive, neighbors) if neighbors > self.survive_max => Cell::Dead,
                    (Cell::Alive, _) => Cell::Alive,
                    (Cell::Dead, neighbors) if neighbors < self.birth_min => Cell::Dead,
                    (Cell::Dead, neighbors) if neighbors > self.birth_max => Cell::Dead,
                    (Cell::Dead, _) => Cell::Alive,
                };
                next.push(next_cell);
            }
        }
        self.cells = next;
    }

    pub fn print(&mut self) {
        for row in 0..self.height {
            for col in 0..self.width {
                match self.get(row, col) {
                    Cell::Alive => print!(""),
                    Cell::Dead => print!(""),
                };
            }
            println!("");
        }
    }
}
```

Con este codigo lo que vamos a hacer es implementar en si las partes interesantes, como calcular el proximo estado de nuestro universo y como mostrarlo por pantalla como lo hicimos en la seccion anterior.

Finalmente vamos por la parte que nos interesa para los diagramas.

```
fn connects_to(id: u32, just_show: bool) {
    let mut cells = vec![Cell::Dead; SIZE];
    for n in 0..SIZE {
        if id >> n & 1 == 1 { cells[SIZE - n - 1] = Cell::Alive }
    }

    let mut universe = Universe {
        cells,
        width: WIDTH,
        height: HEIGHT,
        survive_min: 2,
        survive_max: 3,
    }
```

```
        birth_min: 3,
        birth_max: 3,
    };

    if just_show {
        println!("{}: {}", id);
        universe.print();
        return;
    }

    universe.tick();
    let mut new_id: u32 = 0;
    for n in 0..SIZE {
        new_id = match universe.cells[SIZE - n - 1] {
            Cell::Alive => new_id | 1 << n,
            Cell::Dead  => new_id | 0 << n,
        };
    }
    println!("G.add_edge({}, {})", id, new_id);
}
```

En esta parte vamos a implementar una función sencilla que dada un entero (una enumeración dada) va a crear un universo en ese estado y va a dejar pasar una unidad de tiempo y nos va a decir en qué estado se encuentra ahora.

Finalmente podemos ver que Rust no es un buen lenguaje para ponernos a graficar, por lo que haremos ayuda de Python para graficar nuestro diagrama.

Para hacerlo haremos que Rust cree el script en python

```
fn main() {
    println!("import networkx as nx");
    println!("from netwulf import visualize\n");
    println!("G = nx.Graph()");
}

let limit = 1 << SIZE;
for i in 0..limit {
    connects_to(i, false);
}

println!("\nvisualize(G)");
}
```

Y ahora podemos ejecutarlo con python y ver como se.

Capítulo 3

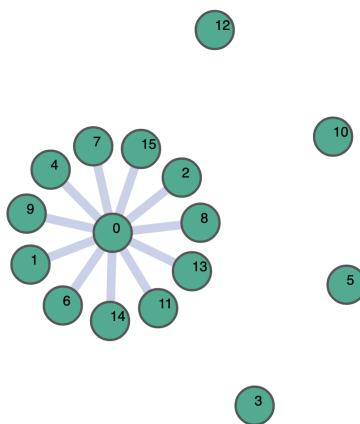
El Juego de la vida: R(2, 3, 3, 3)

3.1. 2x2

El juego de la vida en 2x2 es un muy buen inicio: Al tener solo 4 celulas podemos saber rápidamente que hay 16 posibles estados del universo, podemos ver rápidamente a qué estado va cada otro:

```
(0, 0)
(1, 0)
(2, 0)
(3, 3)
(4, 0)
(5, 5)
(6, 0)
(7, 0)
(8, 0)
(9, 0)
(10, 10)
(11, 0)
(12, 12)
(13, 0)
(14, 0)
(15, 0)
```

Y podemos ver el diagrama final:



Nota que por la biblioteca que usamos si un nodo no tiene arista lo que quiere decir es que esta conectado a si mismo.

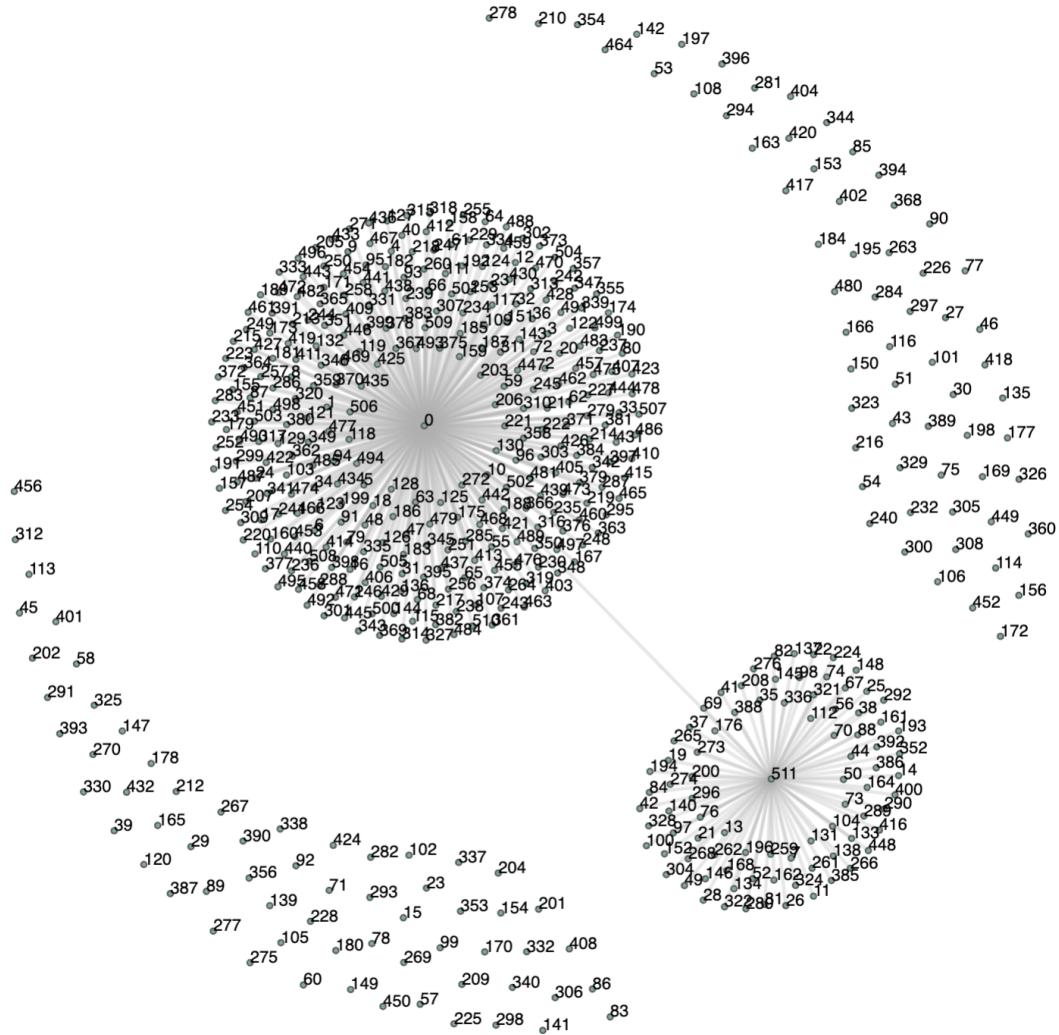
Con el diagrama a la vista podemos ver varios detalles:

- Por un lado el 0 es un gran atractor, de hecho casi todos se van con el
- Y el cero obviamente se queda en cero
- Tenemos otros 4 estados que son estacionarios, es decir que nunca van a cambiar.

3.2. 3x3

El juego de la vida en 3x3 es de mis favoritos, eso si ya tien 512 estados el diagrama, por lo que senti como un gasto no necesario escribirlos pero es muy sencillo hacerlo.

Lo que podemos ver mejor es el diagrama final:



Nota que por la biblioteca que usamos si un nodo no tiene arista lo que quiere decir es que esta conectado a si mismo.

Con el diagrama a la vista podemos ver varios detalles, este ya se puso mucho mas interesante:

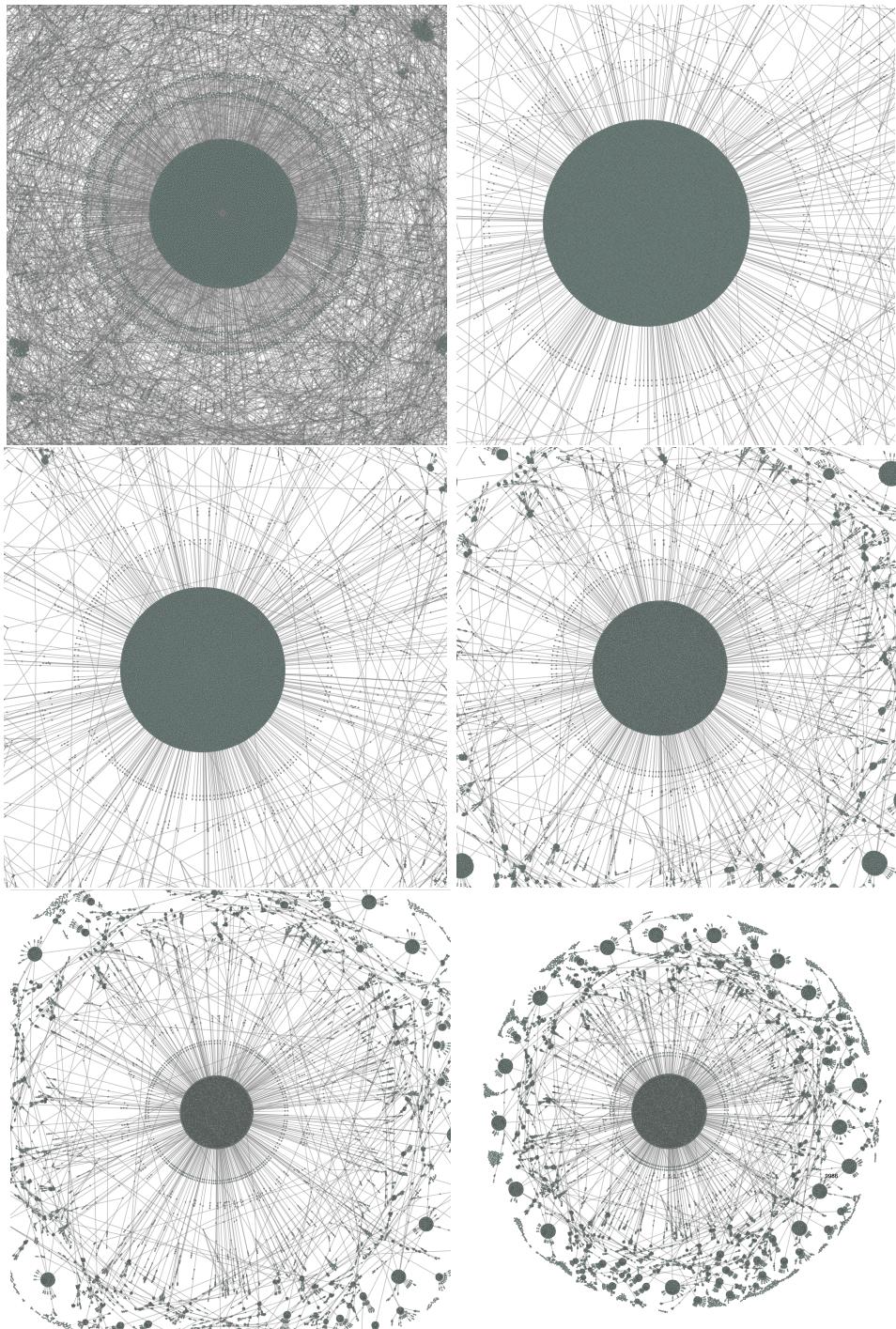
- Por un lado el 0 es un gran atractor, de hecho casi la mitad decaen en el
- Por otro lado ahora tenemos otro atractor, el 511, es decir tener todas celdas encendidas, muchos de los estados van hacia el
- Algo interesante es que el 511 decae al 0
- Finalmente tenemos muchos estados estacionarios, es decir que permanecen en el estado que tenian de manera indefinida.

3.3. 4x4

El juego de la vida en 4x4 es de los diagramas mas difíciles que me toco hacer, de hecho fue por este que me acabe decantando por la libreria que uso para graficar, porque es la unica que lo soportaba.

Lo que podemos fragmentos del diagrama final:

Fue lo mejor que se pudo hacer, pero por lo poco que se puede ver, vemos que tenemos igualmente al cero como el gran maximo, pero que a su alrededor hay varios cumulos de nodos.



3.4. 5x5

Lo intente de mil maneras, incluyendo a Google Colab y Wolfram Cloud, y si bien eran capaces de crear el diagrama, habia tantos nodos que era imposible sacar algo de informacion del mismo.

Capítulo 4

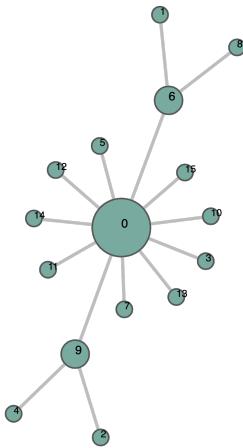
Difusión: $R(7, 7, 2, 2)$

4.1. 2x2

igual es pequeño y podemos escribir a mano los nodos del diagrama

```
(0, 0)
(1, 6)
(2, 9)
(3, 0)
(4, 9)
(5, 0)
(6, 0)
(7, 0)
(8, 6)
(9, 0)
(10, 0)
(11, 0)
(12, 0)
(13, 0)
(14, 0)
(15, 0)
```

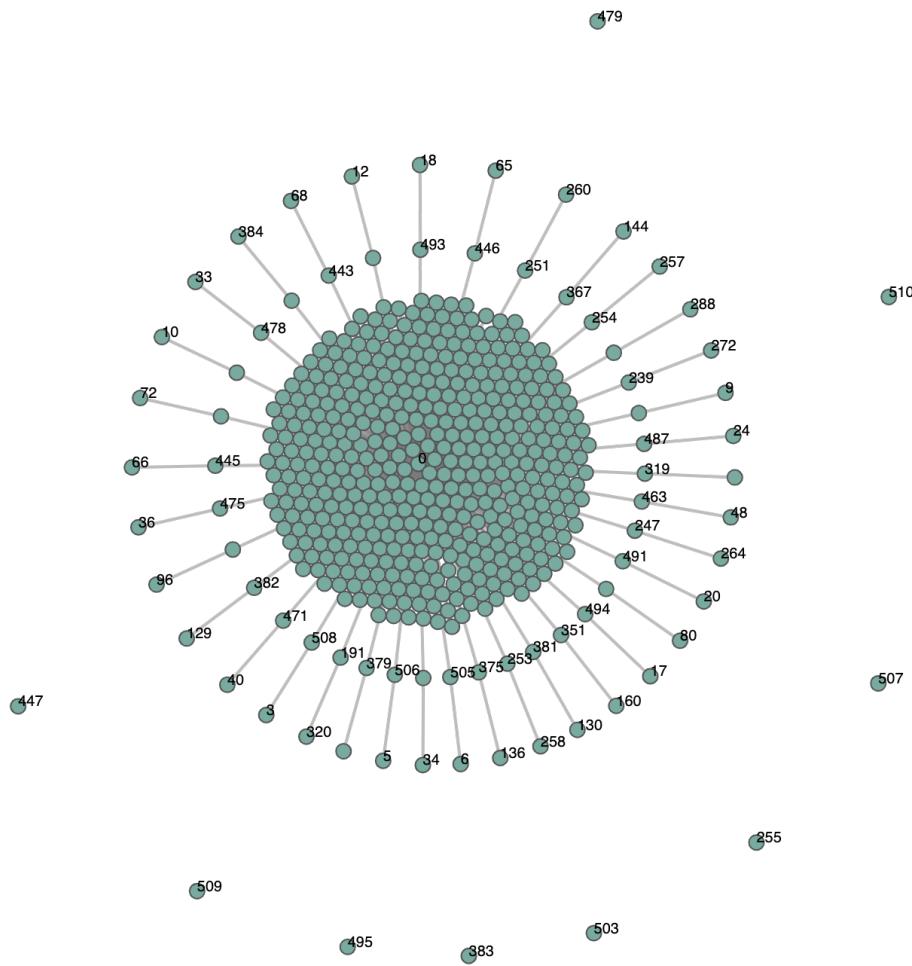
Y podemos ver el diagrama final:



Con el diagrama a la vista podemos ver varios detalles:

- Por un lado el 0 es un gran atractor, de hecho casi todos se van con el
- Y el cero obviamente se queda en cero
- El 9 es el resultado el estado 4, 2
- El 9 cae tambien al cero :c
- El 6 es el resultado el estado 8, 1
- El 6 cae tambien al cero :c

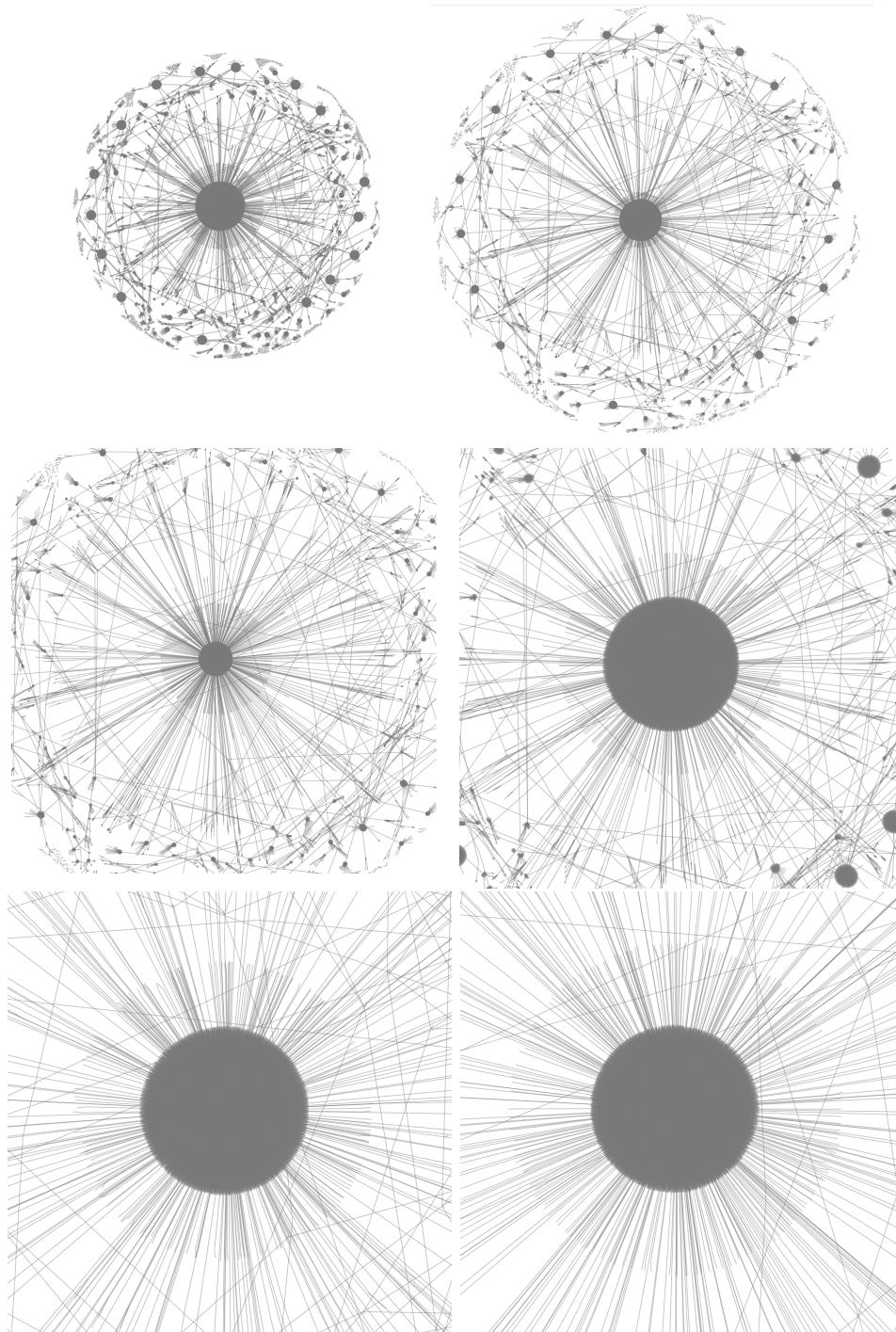
4.2. 3x3



- El 0 sigue siendo el gran atractar
 - Hay una serie de estados que son estaticos, estos son: 255, 383, 447, 479, 495, 503, 507, 509, 510.
 - Hay tambien una serie de nodos que no caen directamente a cero, sino que caen a un estado intermedio y luego a cero.
 - Todos los demas caen a cero en $t+1$.

4.3. 4x4

El 4x4 me dejo sin palabras, solo mira:



Cuenta complejidad.

4.4. 5x5

Lo intente de mil maneras, incluyendo a Google Colab y Wolfram Cloud, y si bien eran capaces de crear el diagrama, habia tantos nodos que era imposible sacar algo de informacion del mismo.

Bibliografía

- [1] *Cellular Automata*. Jarkko Kari, Spring 2013
<https://www.cs.tau.ac.il/~nachumd/models/CA.pdf>
- [2] *A New Kind of Science*. Wolfram Stephen, 2002
- [3] *Introducción a los automatas celulares elementales*. Carlos Zacarias Reyes Martínez
Escuela Superior de Computo