Práctica 8: Autómatas Celulares Implementación paralela.

Karina Flores García y Gabriel Alejandro Herrera Gandarela.

***Abstract*— In this practice we will implement an algorithm for the processing of a matrix by means of cellular automata in the programming language c.**

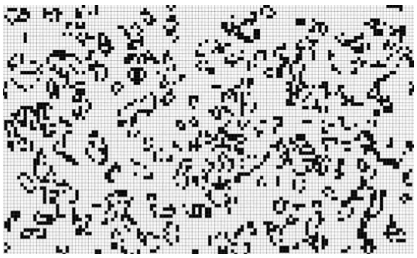
I. OBJETIVOS.

- Procesar una matriz haciendo uso de autómatas celulares.
- Comprender y realizar la implementación de un atómata celular en el lenguaje C.
- Visualizar el autómata con un mapa de calor.
- Realizar la implementación paralela.

II. INTRODUCCIÓN.

Autómatas celulares.

Los autómatas celulares son sistemas dinámicos discretos que se definen de una manera sencilla y poseen una gran complejidad, aparte de muchas otras bondades. Nos interesa presentar su calidad de instrumento relativamente sencillo para modelar sistemas complejos como gases fuera de equilibrio, ferromagnetos, reacciones químicas, sistemas inmunológicos, interacción entre genes biológicos, y ácidos nucleicos. Como veremos, un autómata celular es esencialmente una regla de evolución temporal sobre un conjunto discreto, es decir, dado un punto de este conjunto, presentamos una regla que le asocia otro punto del conjunto. El estudio de sistemas cuya construcción es sencilla pero cuyo comportamiento resulta extremadamente complicado ha llevado a una nueva disciplina de estudio llamada teoría de sistemas complejos, en la cual los métodos computacionales juegan un papel central.[1]
Un ejemplo de ellos, es el juego de la vida:



III. ANÁLISIS DEL PROBLEMA.

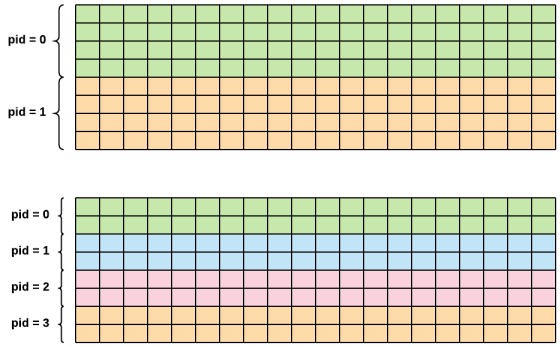
Dada una población, por decirlo de alguna forma, necesitamos generar un programa capaz de identificar todas las formas posibles que se pueden dar. Para empezar analizaremos una secuencia se nos asignó y sobre eso, empezar a observar las múltiples formas que pueden ocurrir, es decir, ir de lo particular a lo general. De primera instancia, es sencillo, sin embargo, lo complicado es que debemos obtener las múltiples formas que pueden ocurrir, entonces, debemos de llevar la lógica adecuadamente para que el objetivo de la práctica se cumpla. Una vez que nuestro programa funcioné de manera correcta, debemos de pasarlo a un mapa de calor para que los resultados se visualicen con mayor facilidad.

IV. IMPLEMENTACIÓN

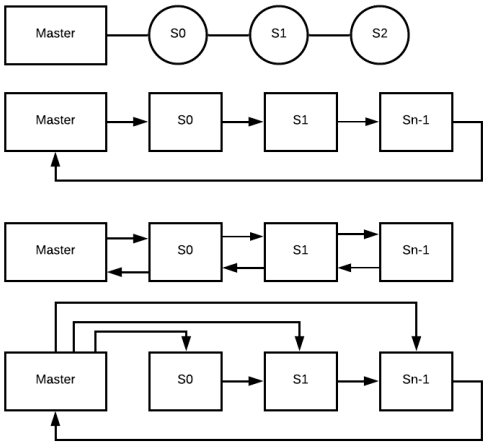
A. Desarrollo

Partiendo de la matriz inicial donde los valores inician con 5 y 0 los valores que se ignorarán, esto porque en el mapa de calor, es más visible dicho rango. Tendremos un iterador, este se encargará de modificar conforme encuentra nuevas secuencias, por ejemplo, cada vez que encuentra una nueva secuencia, incrementará en 1 para que tome valores distintos cada secuencia. Con esta lógica, debemos recorrer la matriz, cada vez que encuentre un valor distinto de cero, entonces cada vez que encuentre un valor distinto de 0, entonces, evaluará su posición y de acuerdo en dónde se encuentra, hará cierta acción. Es decir, habrá fronteras para evitar el desbordamiento, la primera es el nodo inicial, luego es la primera fila, la última fila, la primera columna, la última columna, último valor de la primer fila, primer valor de la última fila, el último nodo y el nodo central, el cual, tiene 8 vecinos a su alrededor.
La lógica consta de si encuentra que sus vecinos anteriores a él, son igual a cero, signica que hay una nueva secuencia, por lo que el iterador aumenta en 1, en caso contrario, si encuentra un valor distinto de 0 y 5, tomará el nodo dicho valor, si encuentra un valor igual a 5, tomará el valor que tiene el iterador. Esto se hará con cada nodo hasta llegar al último, con la condición de sus fronteras.
Para la parte paralela desarrollamos una arquitectura de pipline en donde el procesador 0 se encargaba de repartirle a los demás las filas que debían procesar y este mismo pid procesa las primeras filas que le corresponden. Para repartir las filas, dividimos la cantidad total entre el número de procesadores. Una vez que el procesador que está trabajando ha terminado, le pasa la matriz completa al siguiente procesador.

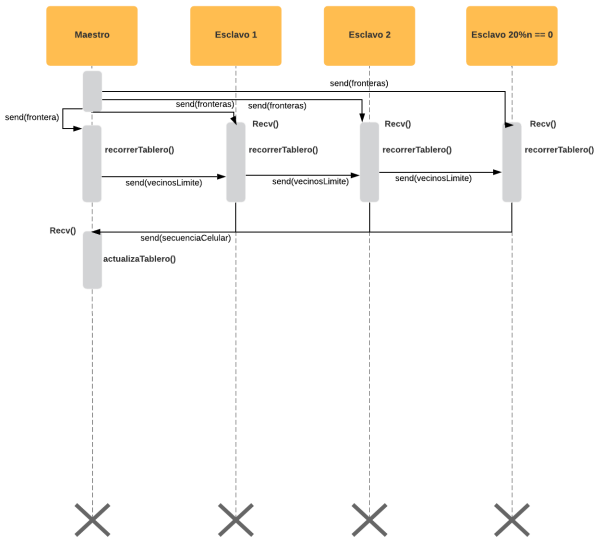
V1	V2	V3
V8	Actual	V4
V7	V6	V5



B. Diagrama de patrón arquitectónico



C. Diagrama UML de secuencia



D. Flujo del programa

V. CÓDIGO FUENTE

```
1  void actualizaVecino(int matriz[8][20], int i, int
       j,int label){
2      //C digo desarrollado para la parte secuencial
        que actualiza un vecino cuando vecinos() lo
       solicita
3  }
4
5  void vecinos(int matriz[8][20],int i, int j, int it
       ){
6      //C digo desarrollado para la parte secuencial
        que evalua la matriz respecto a las fronteras
7  }
8
9
10
11 int main(int argc, char* argv[]){
12
13     tiempo_inicio = MPI_Wtime();
14     if(pid==0){
15       tag = 0;
16       for(int i=1; i<size; i++){
17           indicesFilas[0] = frontera*i; //donde va
       iniciar
18           indicesFilas[1] = indicesFilas[0] +
       frontera; //donde va a terminar
19           MPI_Send(&indicesFilas,sizeof(
       indicesFilas),MPI_INT,i, tag, MPI_COMM_WORLD);
20       }
21       int iniFila = frontera * pid;
22       int finFila = iniFila + frontera;
23       int valor = 0;
24       for(int i=iniFila;i<finFila;i++){
25         for(int j=0;j<20;j++){
26           valor = tablero[i][j];
27           if(valor!=0){
28             vecinos(tablero, i, j,iterador);
29           }
30         }
31       }
32       MPI_Send(&tablero,tamanio,MPI_INT,1, tag,
       MPI_COMM_WORLD);
33     }
34
35
36     if(pid==ultimo){
37       tag = 0;
38       int valor = 0;
39       MPI_Recv(&indicesFilas,sizeof(indicesFilas),
       MPI_INT,0, tag, MPI_COMM_WORLD, &info);
40       MPI_Recv(&tablero,tamanio,MPI_INT,pid-1, tag,
        MPI_COMM_WORLD, &info);
41       int inicio = indicesFilas[0];
42       int fin = indicesFilas[1];
43       for(int i=inicio;i<fin;i++){
44         for(int j=0;j<20;j++){
45           valor = tablero[i][j];
46           if(valor!=0){
47             vecinos(tablero, i, j,iterador);
48           }
49         }
50       }
51
52       //----------Segunda pasada a la matriz para
       correcciones-------
53       iterador = 1;
54       for(int i=0;i<8;i++){
55         for(int j=0;j<20;j++){
56           valor = tablero[i][j];
57           if(valor!=0){
58             vecinos(tablero, i, j, iterador);
59           }
60         }
61       }
62
63       MPI_Send(&tablero,tamanio,MPI_INT,0, tag,
       MPI_COMM_WORLD);
64
65     }else if(pid==1){
66       MPI_Recv(&indicesFilas,sizeof(indicesFilas),
       MPI_INT,0, tag, MPI_COMM_WORLD, &info);
```

```
67        MPI_Recv(&tablero,tamanio,MPI_INT,pid-1, tag,
          MPI_COMM_WORLD, &info);
68        int inicio = indicesFilas[0];
69        int fin = indicesFilas[1];
70        int valor = 0;
71        for(int i=inicio;i<fin;i++){
72          for(int j=0;j<20;j++){
73            valor = tablero[i][j];
74            if(valor!=0){
75              vecinos(tablero, i, j,iterador);
76            }
77          }
78        }
79        MPI_Send(&tablero,tamanio,MPI_INT,pid+1, tag,
          MPI_COMM_WORLD);
80      }else if(pid == 2){
81        MPI_Recv(&indicesFilas,sizeof(indicesFilas),
          MPI_INT,0, tag, MPI_COMM_WORLD, &info);
82        MPI_Recv(&tablero,tamanio,MPI_INT,pid-1, tag,
          MPI_COMM_WORLD, &info);
83          //se hace el mismo procedimiento que en el
          pid=1
84        MPI_Send(&tablero,tamanio,MPI_INT,pid+1, tag,
          MPI_COMM_WORLD);
85      }else if(pid == 3){
86        MPI_Recv(&indicesFilas,sizeof(indicesFilas),
          MPI_INT,0, tag, MPI_COMM_WORLD, &info);
87        MPI_Recv(&tablero,tamanio,MPI_INT,pid-1, tag,
          MPI_COMM_WORLD, &info);
88          //se hace el mismo procedimiento que en el
          pid=2
89        MPI_Send(&tablero,tamanio,MPI_INT,pid+1, tag,
          MPI_COMM_WORLD);
90      }else if(pid == 4){
91        MPI_Recv(&indicesFilas,sizeof(indicesFilas),
          MPI_INT,0, tag, MPI_COMM_WORLD, &info);
92        MPI_Recv(&tablero,tamanio,MPI_INT,pid-1, tag,
          MPI_COMM_WORLD, &info);
93          //se hace el mismo procedimiento que en el
          pid=3
94        MPI_Send(&tablero,tamanio,MPI_INT,pid+1, tag,
          MPI_COMM_WORLD);
95      }
96
97      MPI_Barrier(MPI_COMM_WORLD);
98      if(pid == 0){
99        int ultimoPid = size - 1;
100       MPI_Recv(&tablero,tamanio,MPI_INT,ultimoPid,
          tag, MPI_COMM_WORLD, &info);
101       printf("\n");
102     //Imprime matriz en pantalla
103 //------------------------------
104       if(archivo == NULL){
105         printf("Error en la apertura del archivo\n"
          );
106       }else{
107         //Se imprime la matriz procesada en el
          archivo
108     }
109
110     tiempo_fin = MPI_Wtime();
111     tiempo_total = tiempo_fin - tiempo_inicio;
112     printf("El tiempo de ejecuci n es: %f\n",
          tiempo_total);
113
114     fclose(archivo);
115
116     }
117
118   MPI_Finalize();
119   return 0;
120 }
```

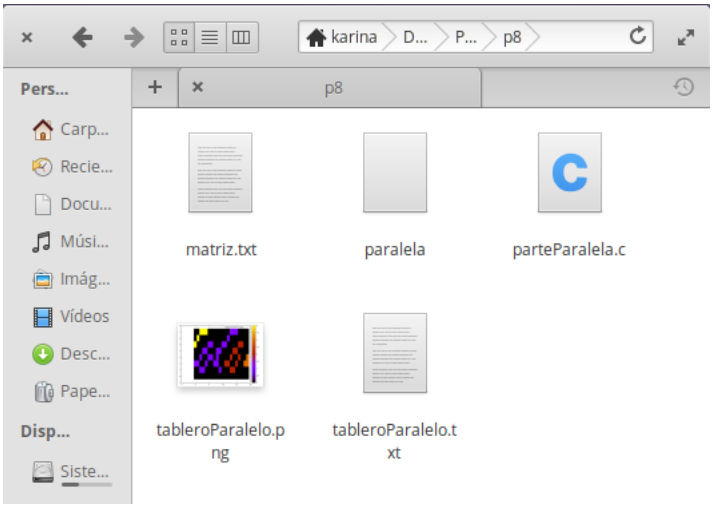
Programa con 2 procesadores.

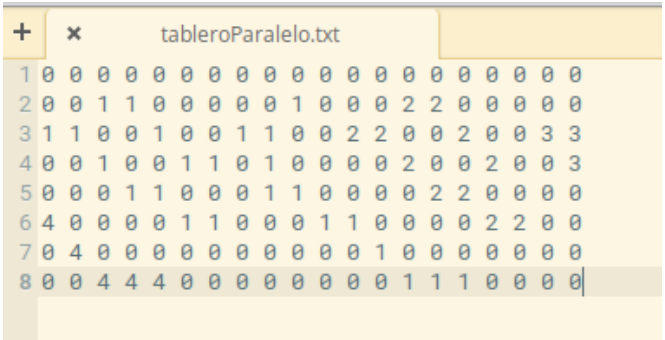
```
karina@karina-VirtualBox:~/Documentos/Practicas_SD/p8$ mpicc partePa
la.c -o paralela
karina@karina-VirtualBox:~/Documentos/Practicas_SD/p8$ mpirun -np 2
lela

0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |
0 |0 |1 |1 |0 |0 |0 |0 |0 |1 |0 |0 |0 |2 |2 |0 |0 |0 |0 |0 |
1 |1 |0 |0 |1 |0 |0 |1 |1 |0 |0 |2 |2 |0 |0 |2 |0 |0 |3 |3 |
0 |0 |1 |0 |0 |1 |1 |0 |1 |0 |0 |0 |0 |2 |0 |0 |2 |0 |0 |3 |
0 |0 |0 |1 |1 |0 |0 |0 |1 |1 |0 |0 |0 |0 |2 |2 |0 |0 |0 |0 |
4 |0 |0 |0 |0 |1 |1 |0 |0 |0 |1 |1 |0 |0 |0 |0 |2 |2 |0 |0 |
0 |4 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |1 |0 |0 |0 |0 |0 |0 |0 |
0 |0 |4 |4 |4 |0 |0 |0 |0 |0 |0 |0 |0 |1 |1 |1 |0 |0 |0 |0 |
El tiempo de ejecución es: 0.001249
karina@karina-VirtualBox:~/Documentos/Practicas_SD/p8$
```

A. *Programa con 4 procesadores.*

```
karina@karina-VirtualBox:~/Documentos/Practicas_SD/p8$ mpirun -np 4
lela

0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |
0 |0 |1 |1 |0 |0 |0 |0 |0 |1 |0 |0 |0 |2 |2 |0 |0 |0 |0 |0 |
1 |1 |0 |0 |1 |0 |0 |1 |1 |0 |0 |2 |2 |0 |0 |2 |0 |0 |3 |3 |
0 |0 |1 |0 |0 |1 |1 |0 |1 |0 |0 |0 |0 |2 |0 |0 |2 |0 |0 |3 |
0 |0 |0 |1 |1 |0 |0 |0 |1 |1 |0 |0 |0 |0 |2 |2 |0 |0 |0 |0 |
4 |0 |0 |0 |0 |1 |1 |0 |0 |0 |1 |1 |0 |0 |0 |0 |2 |2 |0 |0 |
0 |4 |0 |0 |0 |0 |0 |0 |0 |0 |0 |0 |1 |0 |0 |0 |0 |0 |0 |0 |
0 |0 |4 |4 |4 |0 |0 |0 |0 |0 |0 |0 |0 |1 |1 |1 |0 |0 |0 |0 |
El tiempo de ejecución es: 0.002103
karina@karina-VirtualBox:~/Documentos/Practicas_SD/p8$
```

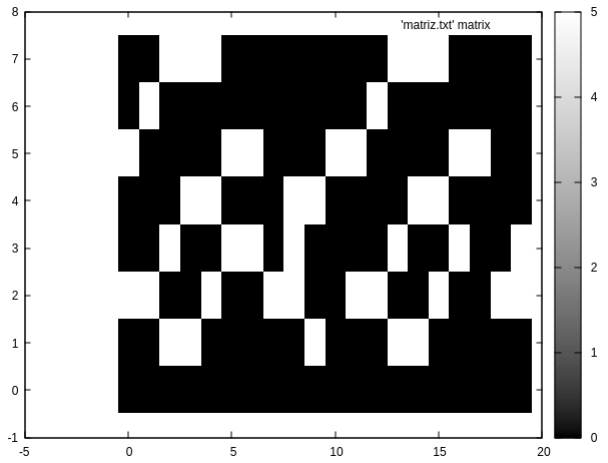
Archivos generados





VI. Mapa de calor

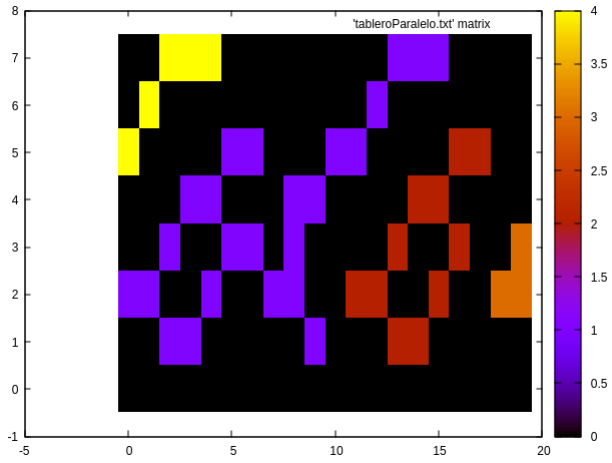
Mapa de la matriz inicial.



Nota: Gnuplot muestra la matriz al revés.

Mapa después del procesamiento paralelamente.

```
karina@karina-VirtualBox:~/Documentos/Practicas_SD/p8$ gnuplot

        G N U P L O T
        Version 5.2 patchlevel 2    last modified 2017-11-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2017
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> plot 'tableroParalelo.txt' matrix with image
gnuplot>
```



Nota: Gnuplot muestra la matriz al revés.

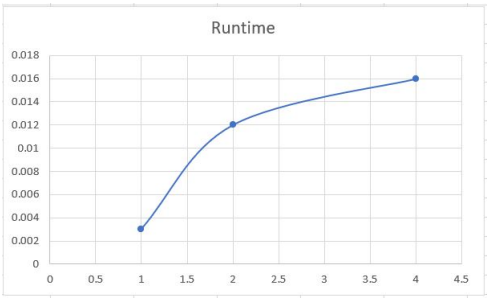
TABLE I

Métrica de rendimiento para *Runtime paralelo*.

Número de procesadores	Runtime (s)
1	0.003
2	0.012
4	0.016

VII. Comparación y métricas.

Runtime Secuencial

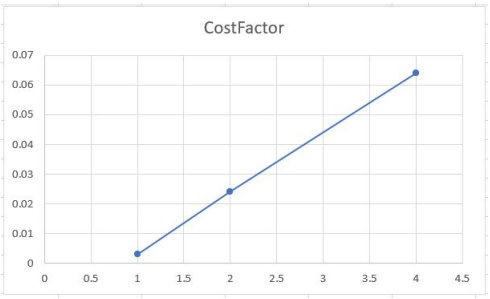


Cost Factor

TABLE II

Métrica de rendimiento para *Costfactor*.

Número de registros	Costfactor (s)
1	0.003
2	0.024
4	0.064



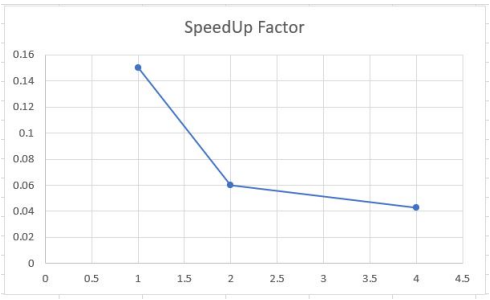
El costfactor aumenta conforme los procesadores aumentan.

Speedup Factor

TABLE III

Métrica de rendimiento para *Speedup Factor*.

Número de procesadores	Speedup Factor (s)
1	0.15
2	0.06
4	0.04

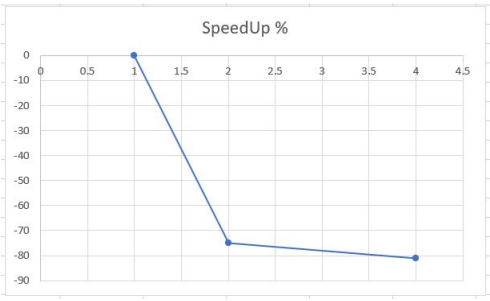


El speedup disminuye conforme los procesadores aumentan.

Speedup Factor %

TABLE IV
MÉTRICA DE RENDIMIENTO PARA *Speedup Factor porcentaje*.

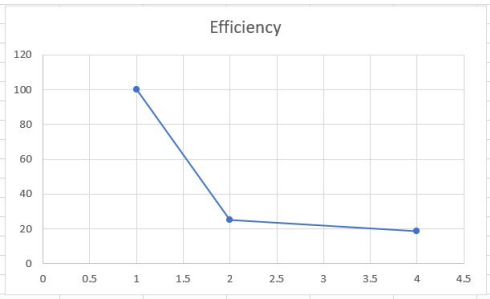
Número de procesadores	Speedup Factor (s)
1	0
2	-75
4	-81.25



Efficiency

TABLE V
MÉTRICA DE RENDIMIENTO PARA *Efficiency*.

Número de procesadores	Efficiency (s)
1	100
2	25
4	18.75



El efficency disminuye conforme los procesadores aumentan.

VIII. CONCLUSIONES

Para el procesamiento de la matriz fue de vital importancia tomar en cuenta los diferentes casos que se presentan en cada elemento. Tuvimos que hacer validaciones debido a que las fronteras de la matriz, es decir la primera y última columna, así como la primera y última fila no presentan los ocho vecinos que los demás elementos sí poseen. Esta fue la parte más laboriosa del algoritmo y en la que se invierte más código.

Logramos la implementación de un atómata celular y pudimos visualizar su comportamiento con un mapa de calor realizado con gnuplot de Linux.

Para el caso del programa en paralelo utilizamos el patrón arquitectónico de pipline para asegurarnos que cuando un procesador hiciera la conversión de los números en su pedazo, la parte izquierda a ese pedazo ya estuviera procesada y pudiera seguir la secuencia para el cambio de iterador y marcar correctamente cada casilla unicamente si tenía vecinos arriba, abajo o a la izquierda. Para corregir los pequeños errores de secuencia que pudieran sucitar el pid = 0 siempre se encarga de recorrer una vez más la matriz y realizar la verificación de las secuencias. Lo anterior fue establecido para la parte secuencial y la paralela.

La dificultad de esta práctica se centra en elegir la forma de llevar a cabo la parte paralela debido a que la condición de carrera en algunas implementaciones no permitían obtener la secuencia adecuada y por ende obteniamos un resultado final incorrecto.

REFERENCES

[1] Rechtman, R. Una introducción a autómatas celulares. Revista Ciencias, UNAM. https://www.revistaciencias.unam.mx/pt/172-revistas/revista-ciencias-24/1572-una-introducci%C3%B3n-a-aut%C3%B3matas-celulares.html

[2] Ayala,J.A.(22 abril 2020) Práctica 8:Autómatas Celulares. Sistemas Distribuidos, 1.