



**Universidad de  
los Andes**  
Colombia

**Facultad de  
Ingeniería**

Universidad de los Andes  
Ingeniería de Sistemas y Computación  
Infraestructura Computacional

## **Documento Caso 1**

Maria Luisa Rodríguez Castillo  
202121549

Danny Camilo Muñoz Sanabria  
202215511

María Paula Alméciga Moreno  
202023369

Bogotá, Colombia  
Febrero 2024

# Contents

## Contents

i

<b>1</b>	<b>Diseño y Funcionamiento del Programa</b>	<b>1</b>
1.1	Diagrama de Clases . . . . .	1
1.1.1	Main . . . . .	1
1.1.2	Matrix . . . . .	2
1.1.3	Buffer . . . . .	2
1.1.4	CellProductor . . . . .	2
1.1.5	CellConsumer . . . . .	2
1.1.6	Implementación del Encuentro . . . . .	3
1.2	Explicación del Funcionamiento Global del Sistema . . . . .	3
1.3	Interacción entre cada pareja de objetos . . . . .	5
1.3.1	Matrix - CellConsumer . . . . .	5
1.3.2	CellConsumer - Buffer . . . . .	6
1.3.3	CellProductor - Buffer . . . . .	7
1.4	Funcionamiento de las barreras . . . . .	7
1.4.1	Barrera 1 . . . . .	7
1.4.2	Barrera 2 . . . . .	7
1.5	Implementación de la sincronización . . . . .	8
<b>2</b>	<b>Validación del programa</b>	<b>10</b>
2.1	Mecanismo para la realización de pruebas . . . . .	10
2.2	Pruebas realizadas . . . . .	11
2.2.1	Matriz 3x3, 4 generaciones . . . . .	11
2.2.2	Matriz 4x4, 5 generaciones . . . . .	11
2.2.3	Matriz 5x5, 7 generaciones . . . . .	12

# Diseño y Funcionamiento del Programa

## 1.1 Diagrama de Clases

Para solucionar el problema propuesto en el Caso, se implementaron las siguientes clases: **Main**, **Matrix**, **Buffer**, **CellConsumer** y **CellProductor**, cuyas relaciones son evidenciadas en el siguiente diagrama de clases:

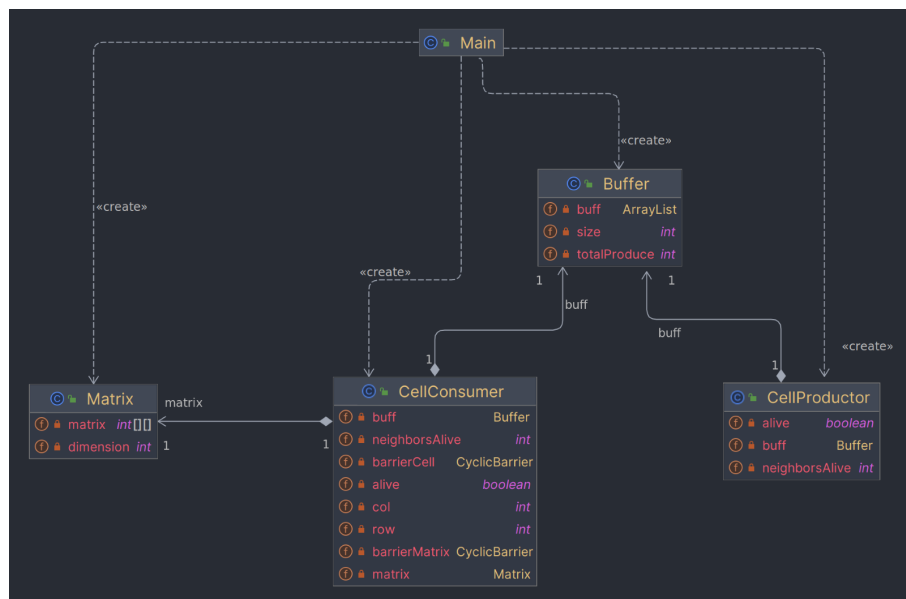


FIGURE 1.1: Diagrama de clases

### 1.1.1 Main

La clase **Main** es la encargada de recibir, mediante la entrada del usuario, un número entero que representará la dimensión de la matriz cuadrada. Luego de recibir este dato, se genera un primer tablero inicial sobre el cual se desarrollarán las futuras generaciones. Por cada celda se crea un

buzón, se genera una lista de sus vecinos, se crea una lista de productores – en la cual existe un productor por cada vecino – y se crea un consumidor que apunta a su respectivo buffer.

### 1.1.2 Matrix

La clase **Matrix** representa el ‘tablero’ sobre el cual se desarrolla el juego. El método **KnowNeighbors()** permite crea una lista de los posibles vecinos que puede llegar a tener una celda, teniendo en cuenta que las celdas ubicadas en el borde de la matriz tendrán una menor cantidad de vecinos. Por otra parte, el método **getValueMatrix()** permite saber si una celda está viva mediante un booleano que revisa el valor de la misma: en caso de que este sea 1, la celda está viva y por lo tanto retorna true, y en el caso contrario, la celda tiene un valor de 0, lo que indica que está muerta.

### 1.1.3 Buffer

La clase **Buffer** tiene un contador llamado **totalProduce** y una cola llamada **buff** que fue impelentada como **ArrayList**. El método **almacenar(i)** espera hasta que tenga espacio y, cuando se libera, notifica a todos los productores, añadiendo el valor del entero **i** a su lista y sumando 1 a **totalProduce**. Por otra parte, el método **retirar()** espera hasta que la cola deje de estar vacía y avisa a todos los consumidores, restando 1 a **totalProduce**.

### 1.1.4 CellProductor

La clase **CellProductor** implementa un **Thread** por cada vecino de cada celda. Es decir, cada celda tiene un máximo de 8 productores asociados que tiene la responsabilidad de informarle su estado a sus vecinos. Lo anterior mediante el método **iAmProductorAlive()**, que define mediante un booleano si el productor está relacionado con una celda viva, es decir, que la celda tiene un valor de 1, o muerta, con un valor de 0. Esta información es almacenada únicamente si la celda está viva.

### 1.1.5 CellConsumer

La clase **CellConsumer** implementa un **Thread** por cada celda, con la responsabilidad de actualizar su estado de vida según las restricciones planteadas por el enunciado: en caso de que una celda esté viva y tenga entre 1 y 3 vecinos, seguirá viva. Si la cantidad de vecinos se encuentra fuera de este rango, entonces muere. Sin embargo, si una celda está muerta y tiene exactamente 3 vecinos vivos, va a estar viva en la siguiente generación.

### 1.1.6 Implementación del Encuentro

En la solución propuesta, se implementan dos barreras: `barrierUpdateCell` y `barrierUpdateMatrix`. La primera es la encargada de asegurarse que se dé un encuentro entre todos los threads de `CellConsumer`, actualizando consistentemente los estados de vida. Por otro lado, la segunda es la encargada de esperar a que se actualicen los estados de todas las celdas antes de imprimir el resultado por consola, de manera que los cálculos de cada generación sean consistentes con los estados de vida calculados.

## 1.2 Explicación del Funcionamiento Global del Sistema

Inicialmente, se crea una matriz según la entrada especificada, en donde 0 = `False` y 1 = `True`.

```
false true false
true true true
false false false
Tablero de la generacion inicial:
0  1  0
1  1  1
0  0  0
```

FIGURE 1.2: Ejemplo de la entrada y representación de la matriz asociada

Posterior a esto, se crea una matriz de buffers totalmente aparte, con el fin de que cada consumidor (celda actual del recorrido) y sus productores asociados (vecinos de la celda actual recorrida) apunten al mismo buffer. Esto sucede, ya que por cada celda actual que visito, se guarda un buffer en esa misma posición de la matriz de buffers. Por ejemplo, si estoy en la celda actual `[0,0]`, el buffer de esa celda y sus vecinos va a estar guardada en la matriz de buffers en `[0,0]`, con el fin de que tanto productores como consumidores asociados, tengan el mismo buffer dinámico. Cabe aclarar que los buffers se van creando a medida que recorro cada celda de la matriz, por lo que inicialmente la matriz de buffers está llena de nulls.

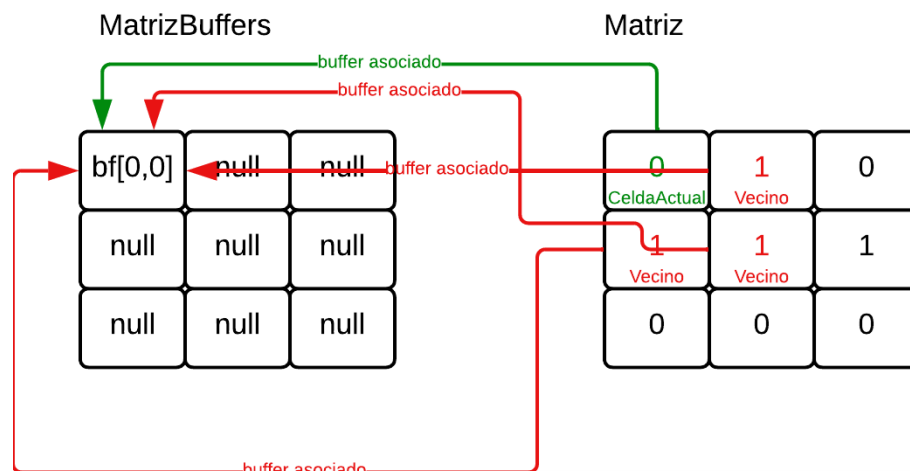


FIGURE 1.3: Asociación de buffers por cada consumidor/productores actuales

Una vez creada la matriz, se crean los threads (productores y consumidores) a medida que se necesiten. Es decir, se inicia el recorrido de la matriz en la posición  $[0,0]$  y se crea un thread consumidor `CellConsumer` asociado a la posición actual. Luego se determina la cantidad de vecinos asociados a esa celda consumidora en  $[0,0]$ . Después de saber cuáles son los vecinos, se crean tantos threads productores como vecinos se han encontrado. Una vez creados, cada uno apunta a su respectivo buffer como se ve en la figura 1.3. El anteriormente proceso descrito se refleja en la figura 1.4.

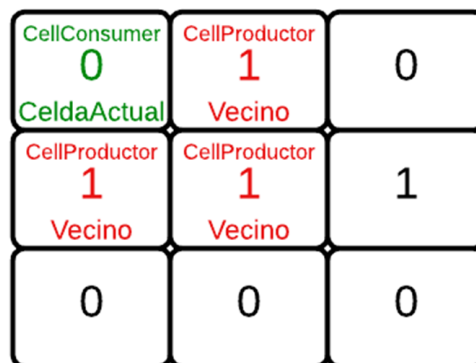


FIGURE 1.4: Asignación de productores y consumidores según la celda actual recorrida

Luego de realizar esta asignación de los consumidores y productores con su respectivo buffer, se utiliza claramente el patrón productor/consumidor. Se decidió usar este patrón debido a la restricción de que recibir y enviar los estados de las celdas debe ser por un buzón, y que el buzón tiene una limitación de tamaño según la fila ( $\text{fila}+1$ ). Además, es fundamental el uso de este patrón, ya que nos permite nivelar el caso en que un thread consumidor consuma más rápido de lo que un thread productor, o viceversa.

Finalmente, cuando se pasan todos los mensajes a cada celda de la matriz NxN con la información de sus vecinos, se hace una barrera con el fin de que la matriz estática asociada a las celdas se actualice

## 1.3 Interacción entre cada pareja de objetos

### 1.3.1 Matrix - CellConsumer

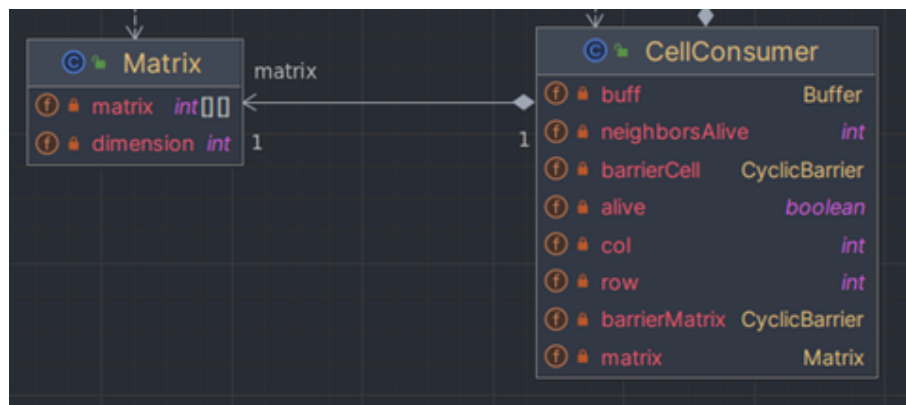


FIGURE 1.5: Relación entre las clases Matrix y CellConsumer

En este caso, cada celda conoce la matriz sobre la cual está trabajando. Cabe resaltar que la clase matriz es estática para todas las instancias de **CellConsumer** en el programa. Esta relación me funciona para que la celda pueda actualizar su nuevo estado en la matriz conforme a la información que se le ha pasado por el buzón, de manera que esta información se actualiza para todas las **CellConsumer** asociadas a esa matriz. Cabe destacar que este comportamiento solo ocurre después de pasar una barrera (las barreras las explicaremos más adelante).

### 1.3.2 CellConsumer - Buffer

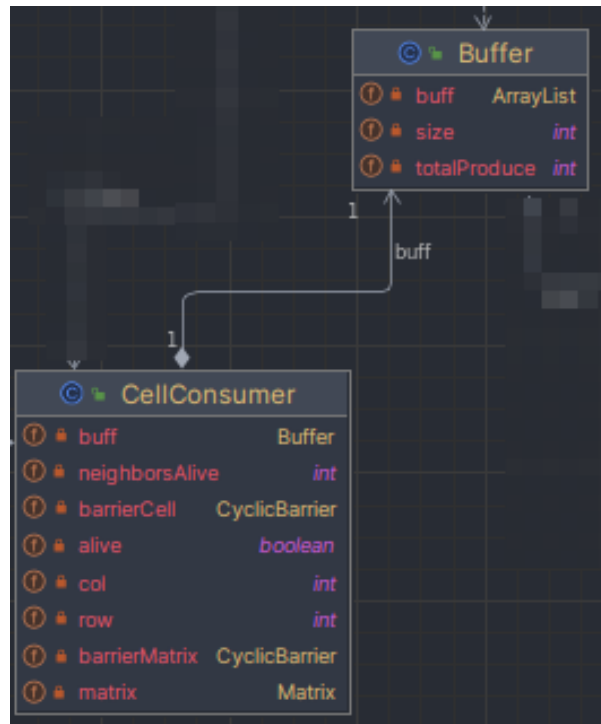


FIGURE 1.6: Relación entre las clases CellConsumer y Buffer

En la implementación por cada celda de la matriz hay un **CellConsumer**, y cada **CellConsumer** tiene su propio buzón. Esta relación funciona para que el consumidor del patrón productor/-consumidor, pueda consumir los datos que el buzón guarda sincronizadamente. Para ser más específicos aun, **CellConsumer** usa el método sincronizado retirar del buffer para consumir los mensajes que contiene.



### 1.3.3 CellProductor - Buffer

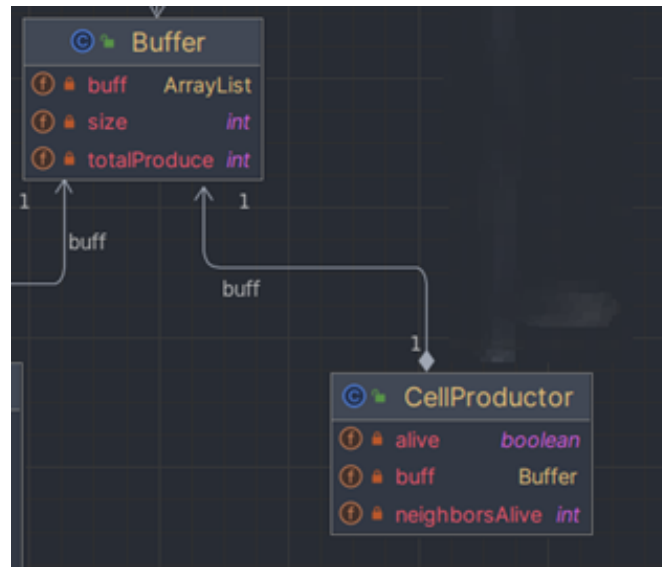


FIGURE 1.7: Relación entre las clases CellProductor y Buffer

En la implementación por cada vecino que hay de una celda actual hay un **CellProductor**, y cada conjunto de **CellProductor** asociados a un consumidor tienen en común su buzón. Esta relación funciona para que el productor del patrón productor/consumidor, pueda enviar los estados de las celdas vecinas sincronizadamente. Para ser más específicos aun, **CellProductor** usa el método sincronizado `almacenar` del `buffer` para almacenar los estados de las celdas vecinas (viva/muerta).

## 1.4 Funcionamiento de las barreras

### 1.4.1 Barrera 1

La primera barrera espera a que todas las celdas consumidoras terminen de consumir sus datos, es decir, la barrera espera que lleguen  $N \times N$  threads. Una vez todas las consumidoras ya hayan terminado de recibir los estados de sus vecinos, todas pasan a actualizar el estado de la matriz de forma concurrente para alistar a la siguiente generación.

### 1.4.2 Barrera 2

La segunda barrera en realidad es opcional, solamente se utiliza para que en el main se pueda imprimir correctamente la siguiente generación justo después de que la matriz ha sido actualizada

por la generación anterior. A fin de cuentas, solo se usa para que imprima correctamente la siguiente generación de la matriz.

## 1.5 Implementación de la sincronización

El buffer funciona como si fuera un monitor. Dicho esto, los métodos que hace el buffer como almacenar o retirar, están sincronizados para favorecer la exclusión mutua, con el fin de que no lleguen dos threads consumidores o productores al mismo segmento de consumir o producir, y se produzcan inconsistencias en los mensajes enviados o recibidos por cada celda.

Para ser más específicos, en el buffer se usan las siguientes funciones:

```
1 public synchronized void almacenar (Integer i) throws InterruptedException {
2     while (this.buff.size() == this.size) {
3         wait();
4     }
5     notifyAll();
6     this.buff.add(i);
7 }
```

Esta función sincronizada utiliza espera pasiva, ya que cuando el buffer está lleno, duerme el thread que llegue hasta que se notifique que el buffer nuevamente está vacío para poder almacenar. Es espera pasiva ya que se duerme hasta que alguien lo despierte y pregunta nuevamente si está lleno o no. En caso de que no esté lleno, simplemente almacena y notifica a todos que pueden almacenar.

Por otro lado se tiene:

```
1 public synchronized Integer retirar () throws InterruptedException {
2     Integer a;
3     if (this.buff.isEmpty()){
4         a = null;
5     }
6     else {
7         a = (Integer) buff.remove (0) ;
8         notify (); ;
9     }
10    return a ;
11 }
```

La función `retirar` en el contexto de un patrón de productores/consumidores, implementa la lógica del consumidor y está sincronizada para evitar run conditions. Si el buffer está vacío, el

consumidor entra en espera semiactiva retornando Null, haciendo que en el `Cellconsumer` se llame el `Yield()` y se ceda el procesador. Por otro lado, cuando el buffer contiene elementos, se retira el primer elemento, notificando a otros hilos. El valor retirado es devuelto como resultado. Además, el hilo consumidor puede implementar una espera semiactiva adicional después de la operación, facilitando una gestión eficiente de recursos, como se describe mediante la implementación de "yield" en el `CellConsumer` que invoca esta función como se mencionó al principio.

# Validación del programa

A continuación se presentan las pruebas de validación llevadas a cabo para rectificar el funcionamiento del programa implementado.

## 2.1 Mecanismo para la realización de pruebas

El programa recibe el tamaño de la matriz y sus valores según la ruta que se le pide al usuario por consola. Por default, se utiliza un archivo *inputs.txt* ubicado en *src/InputCases/inputs* desde la carpeta root del proyecto, que ejecuta en el caso de que el usuario digite erróneamente la ruta del archivo con la información que quiere probar.

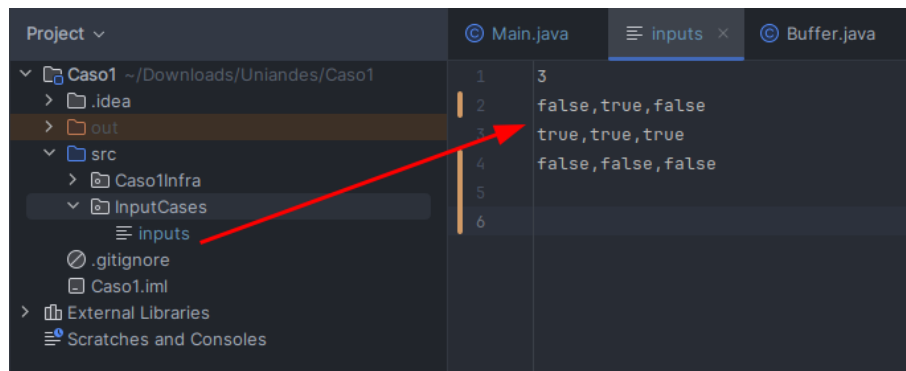


FIGURE 2.1: Vista del archivo de *inputs* con ejemplo de prueba default

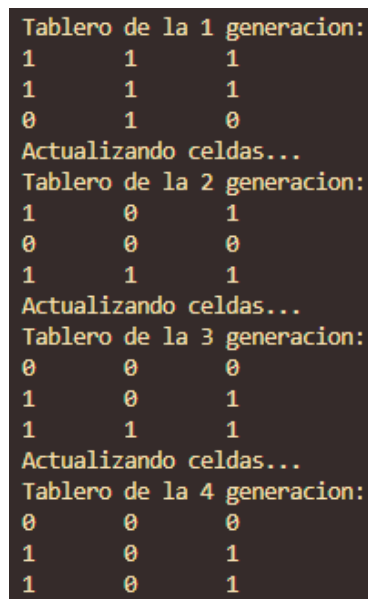
Luego de ingresar los datos relacionados a la matriz, se pide al usuario por consola cuantas generaciones quiere del programa.

## 2.2 Pruebas realizadas

Finalmente, el programa retorna el tablero actualizado de cada generación calculada. Donde los 1 corresponden a celdas vivas, mientras los 0 a celdas muertas. A continuación se muestran las entradas definidas, y el resultado correspondiente para cada generación.

### 2.2.1 Matriz 3x3, 4 generaciones

```
3
false,true,false
true,true,true
false,false,false
```



```
Tablero de la 1 generacion:
1      1      1
1      1      1
0      1      0
Actualizando celdas...
Tablero de la 2 generacion:
1      0      1
0      0      0
1      1      1
Actualizando celdas...
Tablero de la 3 generacion:
0      0      0
1      0      1
1      1      1
Actualizando celdas...
Tablero de la 4 generacion:
0      0      0
1      0      1
1      0      1
```

FIGURE 2.2: Resultado para el primer caso de prueba

### 2.2.2 Matriz 4x4, 5 generaciones

```
4
true,true,false,true
false,false,false,false
true,false,false,false
true,true,false,false
```

```

Tablero de la 1 generacion:
1      1      0      0
1      1      0      0
1      1      0      0
1      1      0      0
Actualizando celdas...
Tablero de la 2 generacion:
1      1      0      0
0      0      1      0
0      0      1      0
1      1      0      0
Actualizando celdas...
Tablero de la 3 generacion:
1      1      0      0
0      0      1      0
0      0      1      0
1      1      0      0
Actualizando celdas...
Tablero de la 4 generacion:
1      1      0      0
0      0      1      0
0      0      1      0
1      1      0      0
Actualizando celdas...
Tablero de la 5 generacion:
1      1      0      0
0      0      1      0
0      0      1      0
1      1      0      0

```

FIGURE 2.3: Resultado para el segundo caso de prueba

### 2.2.3 Matriz 5x5, 7 generaciones

5

```

true,true,false,false,false
true,true,false,false,true
false,false,false,true,false
true,false,true,false,true
true,true,false,true,true

```

```

Tablero de la 1 generacion:
1  1  0  0  0
1  1  1  0  1
1  0  1  1  1
1  0  1  0  1
1  1  1  1  1
Actualizando celdas...
Tablero de la 2 generacion:
1  0  1  0  0
0  0  0  0  1
1  0  0  0  1
1  0  0  0  0
1  0  1  0  1
Actualizando celdas...
Tablero de la 3 generacion:
0  0  0  0  0
0  1  0  1  1
1  0  0  0  1
1  0  0  1  0
1  1  0  0  0
Actualizando celdas...
Tablero de la 4 generacion:
0  0  0  0  0
0  1  0  1  1
1  1  1  0  1
1  0  0  1  0
1  1  0  0  0
Actualizando celdas...
Tablero de la 5 generacion:
0  0  0  0  0
1  1  0  1  1
1  0  0  0  1
0  0  0  1  0
1  1  0  0  0
Actualizando celdas...
Tablero de la 6 generacion:
0  0  0  0  0
1  1  0  1  1
1  1  1  0  1
1  1  0  1  0
1  1  0  0  0
Actualizando celdas...
Tablero de la 7 generacion:
0  0  0  0  0
1  0  0  1  1
0  0  0  0  1
0  0  0  1  0
1  1  1  0  0

```

FIGURE 2.4: Resultado para el tercer caso de prueba