Autómatas celulares 2D en retículas Hexagonales y de Voronoi

Kenny Jesús Flores Huamán
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
kenflohua@alum.us.es

Teodoro Jiménez Lepe
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
teojimenezlepe@hotmail.com

***Resumen*—Los autómatas celulares son herramientas matemáticas y computacionales con la capacidad de modelar gran cantidad de fenómenos. Este trabajo se centra en aquellos con reglas similares al Juego de la Vida, de gran importancia en el campo de la vida artificial.**

Para explorar autómatas sobre mallas no cuadradas, se plantea como objetivo el desarrollo de una herramienta de software. El resultado es el software de código abierto llamado PyCA, programado en Python usando numerosas librerías, entre las que destaca Pygame como biblioteca gráfica para generar las mallas.

Mediante su interfaz de usuario, se generan diversos autómatas celulares sobre malla hexagonal y de diagrama de Voronoi. A partir de estos, se observan y documentan patrones similares a los del Juego de la Vida.

***Palabras clave*—Autómatas celulares, Juego de la vida, Malla hexagonal, Diagrama de Voronoi, Python, Pygame, Tkinter**

I. Introducción

Los autómatas celulares son modelos matemáticos y computacionales para sistemas dinámicos que evolucionan en pasos discretos. Habitualmente se componen de [1]:

- Un conjunto de celdas en el espacio.
- Un conjunto de $\boldsymbol{k}$ estados permitidos para cada celda.
- Un vecindario para cada celda. Este está compuesto por otras $\boldsymbol{m}$ celdas, que influyen en el estado de esta.
- Una regla de transición. Esta hace evolucionar, en cada paso, el estado de cada una de las celdas atendiendo a los estados de sus celdas vecinas.

En la usual malla bidimensional de celdas cuadradas, es común utilizar los vecindarios de de von Neumann ($m = 4$) y de Moore ($m = 8$), que mostramos en la Fig. 1.

Para alterar el vecindario, es posible modificar su rango ($\boldsymbol{r}$). Este hace referencia a la distancia (en la métrica discreta de la malla) a la que una celda ha de estar de otra para que se consideren vecinas. También se puede incluir a cada una de las células en su propio vecindario ($m = 5$ y $m = 9$ respectivamente).

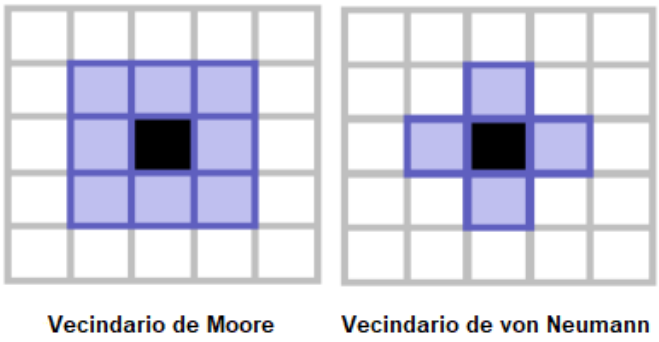


Fig. 1. Vecindarios de autómatas celulares en mallas cuadradas [1]

A partir de la modificación de la geometría de la malla de celdas, el conjunto de estados permitidos, el vecindario y las reglas de transición, es posible configurar una cantidad inmensa de autómatas celulares distintos. Esto permite explorar el conjunto de los autómatas celulares posibles en busca de uno que modele correctamente un sistema dinámico de interés, dando lugar a que los autómatas celulares puedan representar una gran cantidad de fenómenos de múltiples disciplinas [2].

Debido a que las reglas de transición toman como parámetros de entrada la configuración de estados del vecindario de cada célula, la evolución del sistema es gobernada localmente. Sin embargo, estos cambios locales pueden dar lugar a patrones complejos y emergencia de comportamientos colectivos. El comportamiento cualitativo de un autómata celular puede clasificarse en cuatro clases [3]:

- **Clase 1:** Uniforme. La mayoría de los estados iniciales dan lugar a que finalmente todas las células estén en el mismo estado.
- **Clase 2:** Generan patrones periódicos simples.
- **Clase 3:** Se producen comportamientos aparentemente caóticos y no periódicos.
- **Clase 4:** Las reglas producen patrones complejos y estructuras que se propagan por la malla.

Desde el punto de vista de la vida artificial, la complejidad a partir de reglas simples es de gran interés, pues nos permite

hacer una abstracción del concepto de vida y estudiarla tal y como podría ser, y no sólo tal y como la conocemos [4, 5].

A. El Juego de la Vida

Un ejemplo de autómata de gran interés para la vida artificial es el Juego de la Vida. Este autómata celular, diseñado por el matemático John Horton Conway en 1970, está definido sobre una malla bidimensional cuadrada que utiliza el vecindario de Moore ($m = 8$) con distancia $r = 1$. Sus células pueden encontrarse en dos estados: comúnmente se habla de células vivas y muertas. La regla de transición que sigue este autómata es B3/S23:

- Si una célula está muerta:
  - Si tiene exactamente tres vecinos vivos, nacerá.
  - En caso contrario, se mantendrá muerta.
- Si una célula está viva:
  - Si tiene dos o tres vecinos vivos, sobrevivirá.
  - En caso contrario, morirá.

Se trata de una regla de transición que sólo tiene en cuenta el número de vecinos vivos de cada célula, así como el estado de esta misma. Exhibe una gran cantidad de patrones complejos, perteneciendo a los autómatas de clase 4; constituye una máquina de Turing completa.

B. Autómatas hexagonales 2D sobre mallas no cuadradas

La disposición típica de las celdas en el espacio es una malla cuadrada, pero es posible utilizar autómatas celulares que evolucionen sobre teselados triangulares, pentagonales, hexagonales,etc. U otros menos convencionales, como la teselación de Penrose o los diagramas de Voronoi [6, 7].

En este trabajo, nos centraremos en las mallas hexagonales y en los diagramas de Voronoi. Las mallas hexagonales son un teselado del plano utilizando hexágonos regulares, mientras que las mallas de diagrama de Voronoi son un teselado de polígonos no regulares y con diferente número de lados. Otra forma de modificar la geometría de la malla de un autómata es alterar sus condiciones de contorno. En este documento se utilizan condiciones de contorno periódicas para las mallas hexagonales, pero no para los diagramas de Voronoi.

Los vecindarios están estrechamente relacionados con la geometría de la malla sobre la que se define el autómata. En la Fig. 2 se muestran los vecindarios hexagonal ($m = 6$) y de diagrama de Voronoi (número de vecinos distinto para cada celda).

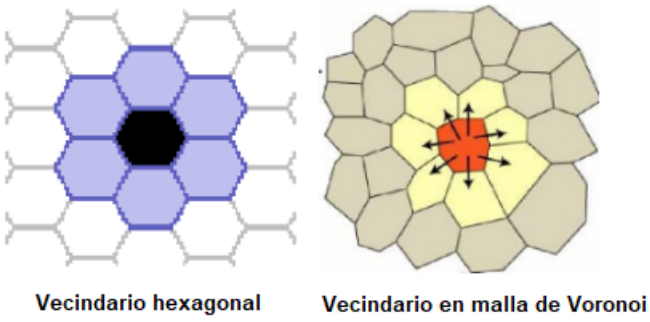


Fig. 2. Vecindarios de autómatas celulares en malla hexagonal y de diagrama de Voronoi [1][2]

Las diferencias en la geometría de la malla, al afectar a las características del vecindario, alteran las propiedades del sistema dinámico que el autómata representa. Esto se debe a que las reglas de transición actúan localmente a través de los propios vecindarios.

Las mallas hexagonales se han utilizado para modelar la propagación de incendios forestales [8], procesos químicos de reacción-difusión [9], flujo de escombros,etc [10]. Además, son particularmente útiles en sistemas dinámicos en los que la reversibilidad es relevante [11].

Es conveniente emplear este tipo de malla cuando el problema estudiado requiera de que todos los vecinos se relacionen entre sí de la misma forma. Esto no ocurre en el vecindario de Moore, en el que algunas relaciones de vecindad se establecen a través de aristas, y otras, a través de vértices.

Por su parte, los diagramas de Voronoi se han utilizado, por ejemplo, para modelar el movimiento de peatones [12], y son adecuados para representar disposiciones de polígonos no regulares en el espacio. Por ello, se plantean como herramientas para modelar sistemas de información geográfica[13].

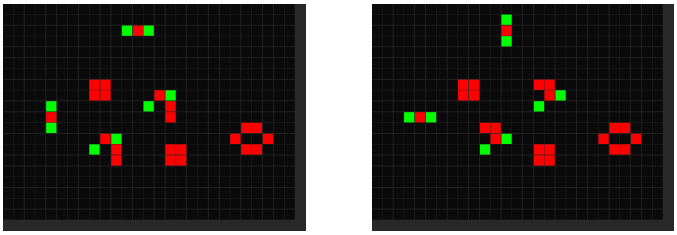
C. PyCA

El objetivo de este trabajo es desarrollar una herramienta de software, *PyCA: Python Cellular Automata*, que permita al usuario simular autómatas celulares de malla cuadrada, hexagonal y de diagrama de Voronoi. Para ello, utilizamos el lenguaje de programación de código abierto *Python*. Concretamente, utilizamos la librería *Pygame* para mostrar los autómatas en pantalla y usar atajos de teclado, y la librería *Tkinter* para la creación de la interfaz gráfica de usuario.

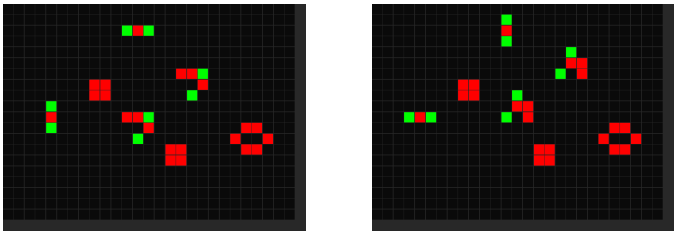
Existen dos motivos para crear esta herramienta, asociados a su vez a dos aplicaciones existentes:

- Brindar una herramienta útil y didáctica para aprender sobre autómatas celulares de una forma similar a la que el software *Netlogo* permite aprender sobre modelos basados en agentes [14].
- Ofrecer la capacidad y el potencial de explotar la diversidad en los autómatas celulares, de la misma forma que el software *Golly* [15] permite una gran capacidad de personalización de autómatas celulares (sobre todo de malla cuadrada).

Nos gustaría plantar una semilla para el desarrollo de una aplicación que permita al usuario utilizar autómatas de geometría personalizada.
En esta versión de la aplicación, sólo implementamos mallas hexagonales, de diagrama de Voronoi, y cuadradas, pero creemos que sería interesante el desarrollo de una herramienta que recopile todas las geometrías exploradas por la comunidad [6, 7].



(a) Iteración 1 (b) Iteración 6



(c) Iteración 11 (d) Iteración 16

Fig. 3. Osciladores, planeadores y vida estática en el Juego de la Vida. Software PyCA.

II. Preliminares

En esta sección se introduce una serie de conceptos básicos que han sido empleados a lo largo de este documento, y son necesarios para que la lectura sea autónoma.

A. *Métodos empleados*

- **Juego de la Vida**
  Este autómata celular presenta patrones de gran interés para la vida artificial. Desde su planteamiento hasta la actualidad se han documentado y clasificado gran cantidad de patrones que son interesantes para abstraer el concepto de vida. Algunos de estos son:
  - **Vidas estáticas:** Estructuras que se mantienen inalteradas.
  - **Osciladores:** Estructuras que pasan por un conjunto de configuraciones periódicamente.
  - **Planeadores:** Estructuras que se propagan por la malla cambiando de forma. Al completar su periodo, se encuentran en otra posición de la malla y con su misma configuración de partida.
  - **Matusalenes:** Estructuras que necesitan de muchas iteraciones para extinguirse o alcanzar un estado en el que sólo existan vidas estáticas y osciladores.
  - **Devoradores:** Estructuras que aniquilan células vivas que se aproximen en determinadas posiciones.
  - **Locomotoras:** Estructuras que se propagan por la malla, como los planeadores, pero dejando un rastro de vida estática.

  En la Fig.3 observamos como evoluciona un entorno en el que cohabitan vidas estáticas, osciladores y planeadores.

  El Juego de la Vida es un aspecto central de este trabajo y de *PyCA*. Las reglas que el usuario puede escoger en la interfaz gráfica de la herramienta desarrollada siguen la estructura que marca este autómata (B3/S23): sólo tienen en cuenta el estado de la célula, y el número de vecinas vivas.
  Además, la mayoría de los experimentos y resultados que se discuten en este trabajo tienen como objetivo encontrar, en malla hexagonal y de Voronoi, patrones de interés para la vida artificial cualitativamente similares a los que presenta el Juego de la Vida sobre malla cuadrada.
- **Simetría de translación**
  Para generar las mallas hexagonales, hemos tenido en cuenta que el espacio que queremos representar es simétrico ante las traslaciones que caracterizan la retícula. Esto quiere decir que teniendo las coordenadas de los vértices de un hexágono, las coordenadas del resto de hexágonos pueden obtenerse sumando a estas un número entero multiplicado por los vectores de traslación de la malla.
- **Diagrama de Voronoi**
  A partir de cualquier conjunto de puntos del plano, denominados semillas, un diagrama de Voronoi es una partición del espacio en regiones que cumplen las siguientes propiedades:
  - Por cada semilla, existe una región.
  - Todos los puntos en el interior de una región están más cerca de su semilla que de cualquier otra del plano.

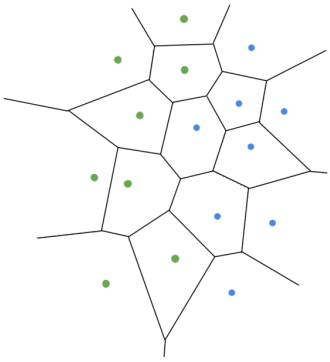


Fig. 4. Diagrama de Voronoi

- **Generación de números aleatorios**
  Los diagramas de Voronoi pueden construirse a partir de cualquier conjunto de puntos en el plano. En esta aplicación, hemos decidido emplear puntos aleatorios para obtener polígonos no regulares.
  Además, hemos utilizado números aleatorios para construir configuraciones iniciales arbitrarias de células vivas y muertas en los autómatas que ofrece la herramienta de software.
- **Programación Orientada a Objetos (POO)**
  Es un paradigma de programación que organiza el diseño del software en torno a **objetos** en vez de funciones [16]. La estructura de la POO incluye los siguientes conceptos:
  - **Clase:** es una plantilla definida por el usuario para crear los objetos, atributos y métodos.
  - **Objeto:** son instancias de una clase creada con atributos definidos. Pueden corresponder a una entidad abstracta o elemento de la vida real.
  - **Método:** son funciones que determinan el comportamiento de un objeto. Se definen dentro de una clase.
  - **Atributo:** es una característica que se le puede asignar a un objeto. Se define también dentro de la plantilla.

  El hacer uso de este paradigma para implementar nuestro programa ha sido una buena idea por las siguientes razones:
  - Los programas que usan este paradigma son mucho más fáciles de leer y comprender. Esto es gracias a la facilidad que existe en abstraer el problema, permitiendo ocultar los detalles menos importantes.
  - En vista de que las clases se dividen en distintos ficheros, hace que sea mucho más beneficioso en el desarrollo colaborativo.
  - Este paradigma permite la reutilización de código por medio de la creación de clases y objetos. Esto nos permite que ahorremos tiempo a la hora de desarrollar nuestro programa.
  - La POO permite desarrollar programas que son más fáciles de ampliar y modificar, ya que se pueden añadir nuevas clases y métodos sin tener que modificar el código existente.
- **Cajas de colisión**
  En la librería *Pygame*, generar botones rectangulares es sencillo, por lo que una malla rectangular cuyas células respondan a los clicks del usuario es fácil de implementar. Sin embargo, las células hexagonales y las regiones poligonales de los diagramas de Voronoi no tienen por qué adaptarse bien a botones rectangulares. Para solucionar este problema, hemos generado cajas de colisión que se adapten a la geometría de las células, permitiendo que estas respondan satisfactoriamente a la interacción del usuario.
- **Particionado del espacio**
  Para determinar los vecinos de cada celda en las mallas cuadradas y hexagonales, podemos recurrir a una estructura matricial, pero no es tan sencillo en el caso de los diagramas de Voronoi.
  Cada célula tiene un vecindario distinto al del resto de células, y ha de ser almacenado antes de simular el autómata. Una solución sería, para cada celda, examinar el conjunto completo del resto de celdas en busca de aquellas que compartan vértices con esta.
  Esta forma de abordar el problema lleva demasiado tiempo, pues para un diagrama de N celdas se han de realizar N^2 iteraciones. Para evitar un tiempo de cálculo demasiado elevado, hemos dividido el plano en secciones rectangulares, de tal forma que cada celda sólo busca posibles vecinas en su misma región.

B. Trabajos Relacionados

En este apartado, se presentarán algunos trabajos importantes en el contexto en el que se encuadra esta herramienta de software. Omitiremos el propio Juego de la vida, que ya ha sido presentado en la *Introducción* y en *Métodos empleados*.

Comenzamos citando explícitamente los trabajos que han inspirado este proyecto y que han aportado la base teórica necesaria para llevarlo a cabo:

- El artículo “Experiments with a somewhat ”Life-like” hexagonal CA” [17]. En este, Paul Callahan es la primera persona en proponer el Juego de la Vida sobre malla hexagonal.
- El artículo científico “2D Hexagonal Cellular Automata: The Complexity of the Forms” [4]: Se trata de un artículo que utiliza el software Golly para explorar patrones complejos en autómatas celulares hexagonales desde el prisma de la vida artificial.
- El artículo científico “Cellular Automata in Triangular, Pentagonal and Hexagonal Tessellations” [7]: En este se trabaja con autómatas celulares de malla no cuadrada y se muestran patrones similares al del Juego de la Vida.
- El artículo científico “Glider dynamics in 3-value hexagonal cellular automata: The beehive rule” [18]: En este se discuten las propiedades de un autómata celular hexagonal con 3 estados permitidos como modelo computacional.
- El artículo científico “Development of Voronoi-based cellular automatan integrated dynamic model for Geo-

graphical Information Systems" [13]: En este trabajo se discute la aplicación que pueden tener los autómatas celulares en diagramas de Voronoi, y cómo pueden ser beneficiosos en modelos geográficos.
A continuación recopilamos herramientas de software que permiten al usuario experimentar con autómatas celulares de malla hexagonal:

- Golly [15]: Permite generar vecindarios hexagonales a partir de la omisión de algunas celdas en una malla cuadrada.
- *The Hexagonal Game Of Life*: Es una simulación web del Juego de la Vida en mallas hexagonales.
- *Hexlife*: Es otra simulación más simple del Juego de la Vida en un tipo de malla hexagonal. Cuenta con los botones básicos para realizar experimentos.
- *Hex Life: Hexagonal Cellular Automata*: Programa basado en el Juego de la Vida, pero en lugar de tener celdas con 4 vecinos, cada celda puede tener hasta 12 vecinos.

III. Metodología

En esta sección se va a presentar una explicación de los diferentes componentes del software realizado. Para el desarrollo, se han utilizado archivos Python y una serie de recursos que se describirán a continuación:

- **main.py:** Archivo principal que genera y visualiza autómatas celulares. Ese fichero incluye funciones para generar nuevas instancias de las diferentes mallas que soporta (cuadrada, hexagonal, Voronoi) el programa. Además, se ha incorporado una interfaz gráfica para configurar algunos elementos de la simulación como puede ser el tamaño de las células.
- **stage.py:** Archivo con extensión Python que se utiliza como superclase para generar los distintos escenarios (mallas) dentro de nuestro programa. Las clases hijas de *Stage* son:
  - **Square (square.py)**
  - **Hexagon (hexagon.py)**
  - **VoronoiGrid (voronoi.py)**

  Todas ellas heredan métodos y atributos de su clase padre, *Stage*, e incorporan los suyos propios, que las distinguen del resto. Fundamentalmente, son scripts para inicializar, representar y actualizar los autómatas. También permiten gestionar la interacción del usuario con *Pygame* a través del teclado y el ratón.
- **pixel_perfect_polygon_hitbox.py:** Archivo en Python que determina si un punto se encuentra dentro o fuera de un polígono convexo. Esto nos permite obtener cajas de colisión para las celdas de la malla de diagrama de Voronoi (ya sean polígonos regulares o irregulares).
- **requirements.txt** Fichero que especifica las dependencias del software y nos facilita la instalación de ellas.

Una vez visto todos los componentes que forman parte del proyecto, se va a explicar cómo se ha desarrollado el trabajo. El proceso de desarrollo de este proyecto se dividió en las siguientes etapas:

A. Autómatas celulares en diferentes mallas

El primer paso para implementar autómatas celulares es la generación de mallas con las diferentes geometrías de interés. Nuestra primera aproximación consistió en generar un programa simple para simular un autómata hexagonal. Para ello, empleamos una librería llamada *hexallatice* [19]. Sin embargo, esta librería está ideada para ser utilizada con *matplotlib*, una librería para generar gráficos.

Tras diseñar un autómata hexagonal completamente funcional utilizando este método, observamos que no era eficiente, pues la generación de cada fotograma tardaba demasiado. En su lugar, nos decantamos por el uso de *Pygame*. Este, al estar escrito parcialmente en *C* es mucho más rápido.

Desde el momento en el que decidimos usar *Pygame*, todo nuestro trabajo tenía que basarse en representar imágenes en un mapa de 800×600 píxeles (dimensiones escogidas por nosotros). Es decir, trabajamos con coordenadas que toman números naturales.

La librería *Pygame* permite dibujar polígonos cualesquiera a partir de sus vértices, así como almacenar una caja de colisión rectangular que englobe al polígono. Gracias a este recurso, el generar las mallas rectangulares y hexagonales fue sencillo. Los pasos a seguir fueron los siguientes:

- Generar una estructura matricial cuyos elementos representan las celdas de la malla.
- Generar las coordenadas de los vértices del polígono regular correspondiente.
- Crear una función que, dados los vértices del polígono asociado al elemento de matriz $(0, 0)$, genere el polígono asociado al elemento de matriz (i, j).

Esto nos permite generar cualquier malla cuadrada o hexagonal en nuestra ventana de *Pygame*. Sin embargo, no es posible aplicar este esquema a los polígonos de los diagramas de Voronoi. Para este teselado, empleamos la librería *scipy*; concretamente el módulo *scipy.spatial.Voronoi*.

Este nos permite generar un diagrama de Voronoi tomando como datos de entrada un conjunto de puntos en el plano. Entre otras salidas, nos proporciona las regiones de Voronoi (polígonos) etiquetadas con números enteros, así como las coordenadas de sus vértices, también etiquetados con números enteros.

Una vez tenemos acceso a los vértices de los polígonos, ya es viable representar diagramas de Voronoi en *Pygame*.

B. Clases

Una vez determinada la viabilidad de las mallas necesarias, estructuramos el software utilizando el paradigma de la programación orientada a objetos (POO).

El objetivo es diseñar los autómatas cuadrados, hexagonales y de Voronoi de una forma análoga, de tal forma que sus semejanzas sean muchas, y puedan plasmarse en una clase padre de la que todas ellas hereden. Dejamos de hablar, por tanto, de programas independientes que permiten construir autómatas, y comenzamos a tratar con instancias de la superclase a la que denominamos Stage.

Las instancias de la superclase Stage pueden pertenecer a las clases Hexagon, Square o Voronoi. Estas subclases están definidas en los archivos hexagon.py, square.py y voronoi.py respectivamente. Cada uno de estos archivos tiene tres bloques fundamentales:

- Inicialización
- Actualización
- Gestión de eventos

A continuación, describimos cada uno de ellos.

1) ***Inicialización****:* El bloque de iniciación carga los datos de la malla y el autómata celular, así como algunas otras características necesarias para gestionar las interfaces y los eventos que se produzcan. Los procesos que se llevan a cabo son los siguientes:

- *Generación de un array binario*: Este representa los estados de cada una de las células del autómata. Los ceros indican muerte, y los unos, vida. En el caso de las mallas hexagonales y cuadradas, el array toma forma matricial, mientras que en el caso de Vornoi, de vector columna. Los valores iniciales que toma este array se ajustan a una probabilidad que el usuario puede personalizar.
- *Almacenamiento de cajas de colisión*: Para cada polígono, no importa de qué malla se trate, almacenamos las cajas de colisión rectangulares que nos aporta *Pygame*. Estas constituyen el rectángulo de menor área capaz de englobar al completo estos polígonos.
- *Determinación del vecindario de cada celda*: A pesar de que esto es sencillo para las mallas cuadradas y hexagonales, debido a su estructura matricial, resulta complicado para los diagramas de Voronoi, como hemos explicado en la sección de *Métodos empleados*. Es importante precalcular estos datos para evitar ralentizar el autómata durante su ejecución.
- *Representación de la malla inicial*: A partir de sus vértices, podemos mostrar en pantalla los polígonos que conforman el teselado.
- *Otros procesos de inicialización*: Algunos de ellos son la definición de variables antes de su uso, el procesado de los datos que el usuario emite a través de la interfaz de *tkinter*, o el establecimiento de banderas que irán cambiando durante la ejecución del programa para procesar paradas y continuaciones, creación de gifs, etc.

2) ***Actualización****:* En este bloque se especifica cómo evoluciona el sistema tras cada paso en el tiempo. Los procesos ejecutados son los siguientes:

- *Modificación de parámetros personalizados por el usuario a través de la interfaz.*
- *Determinación del número de vecinos vivos de cada célula.*
- *Aplicación de la regla de transición*: El array binario es modificado.
- *Representación de la nueva malla con sus colores modificados.*

3) ***Gestión de eventos****:* En este bloque se programa la interacción del teclado y el ratón con la ventana de *Pygame*. Mostramos el pseudocódigo en la figura 5.

handle_events(V)
Entrada: Atributos de la clase
Salida: Vacío

```
Por cada evento E detectado en Pygame:
    si E es el evento salir entonces
        salir de la aplicación
    si E el usuario pulsa ESPACIO entonces
        pausar/retomar la ejecución continuada
    si E el usuario pulsa DERECHA entonces
        actualizar la malla una vez
    si E el usuario pulsa ABAJO entonces
        matar todas las células
    si E el usuario hace CLICK IZQUIERDO entonces
        Por cada celda C en la matriz
            si POS en la caja de colisión de C
                C vive
    si E el usuario hace CLICK DERECHO entonces
        Por cada celda C en la matriz
            si POS en la caja de colisión de C
                C muere
    si E el usuario hace CLICK RUEDA entonces
        Por cada celda C en la matriz
            si POS en la caja de colisión de C
                se despliega información de C
```

Fig. 5. Pseudocódigo de gestión de eventos

C. Archivo principal

El programa *main.py* importa el resto de archivos del software y ejecuta los métodos necesarios para el funcionamiento de la iniciación y el bucle principal de la aplicación. En la figura 6 mostramos su pseudocódigo.

main.py$()$
Entrada: Vacío
Salida: Vacío

```
iniciar pygame
iniciar interfaz mediante tkinter
generar e iniciar instancia de stage por defecto: hexagon
Mientras verdad
    gestionar eventos globales
    si la instancia de stage está en ejecución
        actualizar la instancia
        avanzar un tick de Pygame
```

Fig. 6. Pseudocódigo del archivo principal

Como se puede observar en el pseudocódigo de main.py, los bloques de inicialización, actualización y gestión de eventos son fundamentales para el funcionamiento del programa. En el bucle principal, se utiliza la acción *gestionar eventos globales*. Esta es una generalización de la gestión de eventos descrita anteriormente: no sólo maneja los eventos del autómata, sino también otros asociados a la comunicación de *Pygame* y *tkinter*.

D. Diseño de la interfaz gráfica de usuario (GUI) para personalizar los parámetros del autómata celular

Después de haber desarrollado la lógica que van a seguir los autómatas celulares en los distintos tipos de mallas, se ha decidido implementar una interfaz gráfica que permita al usuario personalizar los parámetros de un autómata celular de una forma que sea intuitiva.

El motivo principal por el cual se decidió realizar esta GUI es que al tener la capacidad de ajustar los parámetros del autómata de una forma rápida y sencilla, se pueden efectuar muchas más pruebas y experimentos en menos tiempo, mejorando la eficacia y el rendimiento.

Para implementar la interfaz, se decidió hacer uso de la librería *Tkinter* a causa de la enorme documentación que tiene este módulo, como la gran sencillez a la hora de desarrollar GUIs.

En primer lugar, se decidió desarrollar un *mockup* haciendo uso de una herramienta de diseño gráfico, para visualizar cómo podría ser una interfaz intuitiva para nuestra aplicación.

Tras haber elaborado el prototipo, implementamos los diferentes elementos gráficos como botones o menús desplegables usando *Tkinter* como podemos observar en la figura 7. Además, también asociamos eventos a los elementos de la GUI.

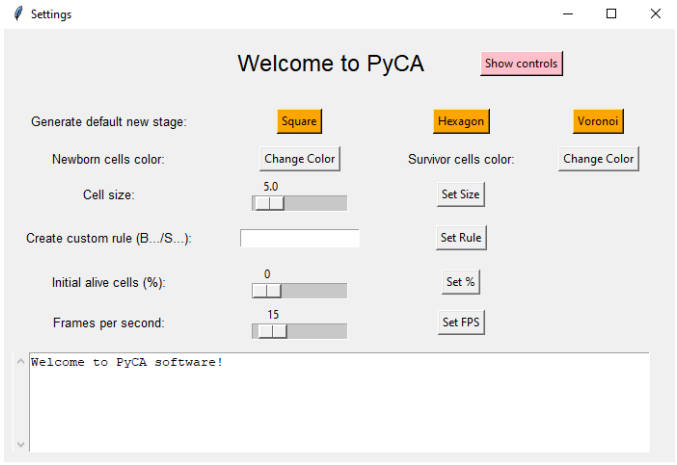


Fig. 7. Interfaz del software PyCA

Finalmente, se realizaron pruebas para verificar que la interfaz funcionaba correctamente.

E. Pruebas y validación del software

El objetivo de este trabajo no era sólo la creación del software, sino también usarlo para aprender acerca del comportamiento de los autómatas celulares de malla hexagonal y de diagrama de Voronoi. Por lo tanto, hemos utilizado la aplicación como cualquier usuario futuro podría. Con cada uso de esta, hemos detectado y resuelto errores iterativamente. El uso personal que le hemos dado al programa nos ha permitido pulir el funcionamiento de la interfaz gráfica y mejorar la eficiencia del autómata celular, obteniendo como resultado una aplicación funcional y útil, sobre todo desde el punto de vista didáctico.

Nuestro uso de la aplicación para obtener los resultados que discutimos en la siguiente sección se ha basado en las siguientes actividades:

- Generación de estados aleatorios.
- Generación de figuras geométricas simétricas.
- Empleo de reglas de transición arbitrarias, similares al Juego de la Vida o encontradas en la literatura.
- Observación de la evolución de estas configuraciones iniciales.
- Recopilación de los patrones emergentes destacables.
- Reproducción de los patrones encontrados en la literatura.

F. Generación de gifs en nuestro programa

Que nuestro software sea capaz de generar gifs dentro de nuestro programa puede ser útil si el usuario quiere visualizar el comportamiento de autómatas celulares a lo largo del tiempo.

Para llevar a cabo esta característica se decidió hacer uso de la librería *imageio*, que proporciona una interfaz para leer y escribir una gran variedad de tipos de imágenes, donde se incluyen las imágenes animadas [20]. Esto es debido a que Pygame no tiene una función específica para generar gifs. En la figura 8, mostramos el pseudocódigo de la gestión de eventos incluyendo esta función adicional.

handle_events(V)
Entrada: Atributos de la clase
Salida: Vacío

```
Por cada evento E detectado en Pygame:
    si E es el evento salir entonces
        salir de la aplicación
    si E es el evento avanzar entonces
        actualiza la malla
    .
    .
    .
    Si E es el evento GRABAR entonces
        esta_grabando ← !esta_grabando
        si esta_grabando entonces
            frames ← Inicializamos buffer de imágenes
            grabar() # Al activar esta función cada vez
            # que actualicemos la malla, se guardará
            # un fotograma en frames
        si esta_grabando y existen frames entonces
            paramos grabación
            frames ← ∅
```

Fig. 8. Pseudocódigo de generación de un gif

G. Documentación de nuestro proyecto en una wiki

En vista que uno los factores clave de nuestro proyecto es que el contenido sea accesible para cualquier persona interesada, así como como para tener una buena plataforma didáctica para exponer todo este contenido aprendido por nosotros,

decidimos crear una *wiki* haciendo uso de la plataforma GitHub para documentar nuestro proyecto software.

Para llevar a cabo la wiki, estuvimos documentándonos sobre como crear y mantener una wiki en GitHub y decidimos seguir los siguientes pasos:

- Se define la estructura y el contenido que va a tener la wiki.
- Se desarrolla el contenido de la *wiki* usando el lenguaje de marcado Markdown.
- Se añaden imágenes para ilustrar el contenido de forma visual.
- Se añaden documentación externa y recursos útiles para ampliar la información.

H. Preparación del ejecutable

Además de la elaboración de este trabajo de investigación y la creación de una wiki, en la última etapa del proyecto se decidió realizar un ejecutable llamado "main.exe".

Para poder generar este archivo utilizamos *Pyinstaller*, una herramienta para crear ejecutables a partir de proyectos escritos en Python [21].

La razón por la cual se decidió elaborar este ejecutable fue con la finalidad de que se pueda asegurar el poder utilizar el programa sin tener que tener instalado Python ni las dependencias. Esto es posible gracias a que la librería *Pyinstaller*, incluye el propio intérprete como las librerías dentro del ejecutable, haciendo que nuestro software sea más fácil de utilizar por otras personas.

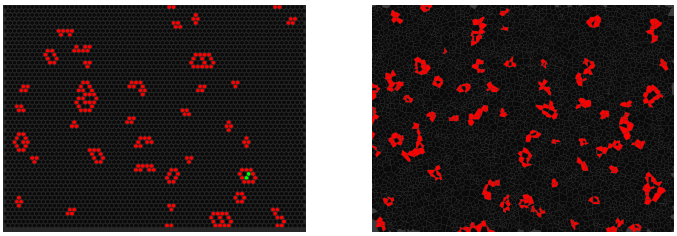
IV. Resultados

Los experimentos realizados tienen como objetivo la determinación de reglas que generen patrones similares a los del Juego de la Vida original. Para ello, seguimos los criterios marcados por Conway [22]:

- No debería haber patrones iniciales cuyo crecimiento descontrolado pueda demostrarse fácilmente.
- Debería haber patrones iniciales que parezcan crecer descontroladamente.
- Debería haber patrones iniciales que evolucionen durante un tiempo considerable antes de decaer a dos configuraciones posibles:
  - Configuración en la que todas las células están muertas.
  - Configuración en la que sólo haya vida estática u osciladores.

Además buscamos autómatas celulares que constituyan ejemplos de las diferentes clases mencionadas en la *Introducción*. Para llevar a cabo esta tarea ejecutamos las acciones descritas en la sección de *Pruebas y validación de software*. Para mejorar la comprensión del autómata, no utilizamos únicamente dos colores. A las células recién nacidas, les asignamos el color verde, y a las supervivientes, el color rojo. Las células muertas, por su parte, siempre tienen color negro.

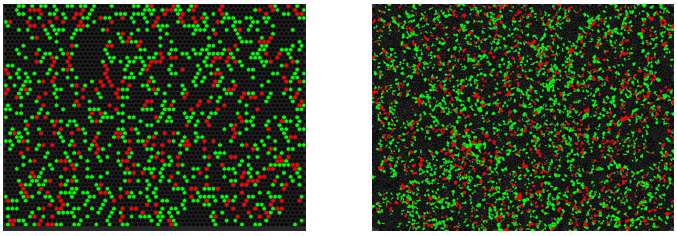
- **B3/S23** La primera regla que ejecutamos para cada una de las tres mallas es la regla original del Juego de la Vida. En la malla cuadrada, hemos observado los característicos patrones del juego de la vida: vida estática, osciladores, planeadores, etc. En las mallas hexagonal y de Voronoi, en cambio, observamos que predomina la vida estática. Mostramos estos resultados en la Fig. 9



(a) (b)

Fig. 9. Autómatas celulares hexagonal (izquierda) y de Voronoi (derecha) con la regla B3/S23

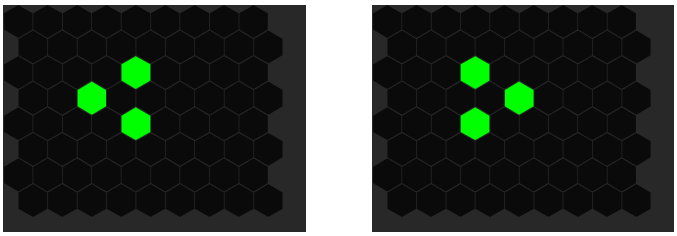
- **B2/S2**: Ante configuraciones iniciales aleatorias, esta regla genera un crecimiento descontrolado en la malla hexagonal y de Voronoi, debido a que el número de células necesarias tanto para nacer como para sobrevivir es relativamente bajo. Mostramos la evolución de un escenario aleatorio tras múltiples iteraciones en la Fig. 10.



(a) a (b) a

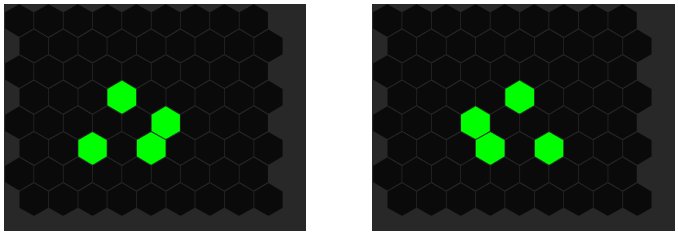
Fig. 10. Autómatas celulares hexagonal (izquierda) y de Voronoi (derecha) con la regla B3/S23

A pesar de esto, algunas configuraciones iniciales con pocas células vivas permiten observar patrones similares a los que se encuentran en el Juego de la Vida: en las Fig. 11,12,13 mostramos varios osciladores, y en la Fig. 14, un planeador.



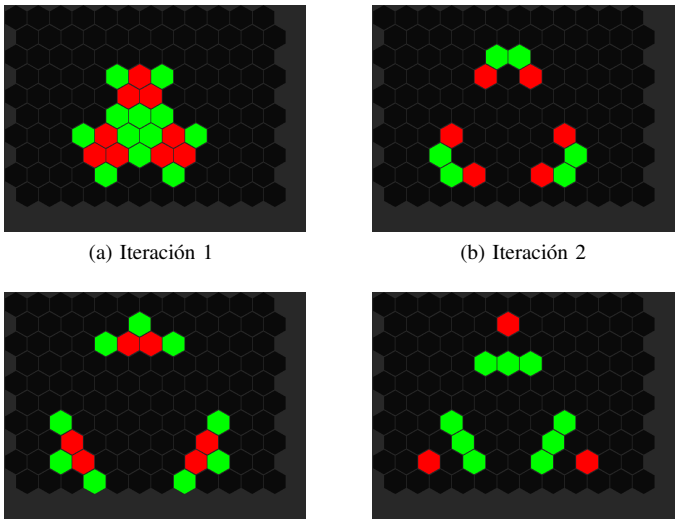
(a) Iteración 1 (b) Iteración 2

Fig. 11. (B2/S2) Oscilador de periodo 1



(a) Iteración 1 (b) Iteración 2

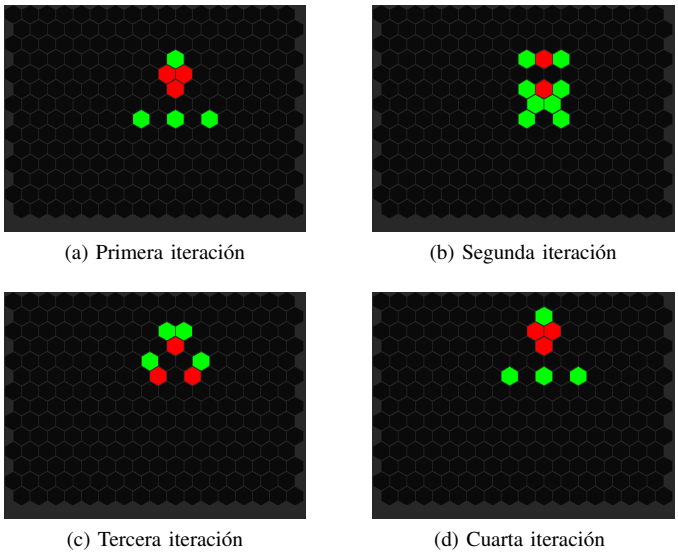
Fig. 12. (B2/S2) Oscilador de periodo 1



(a) Iteración 1 (b) Iteración 2

(c) Iteración 3 (d) Iteración 4

Fig. 13. (B2S2) Oscilador de periodo 3

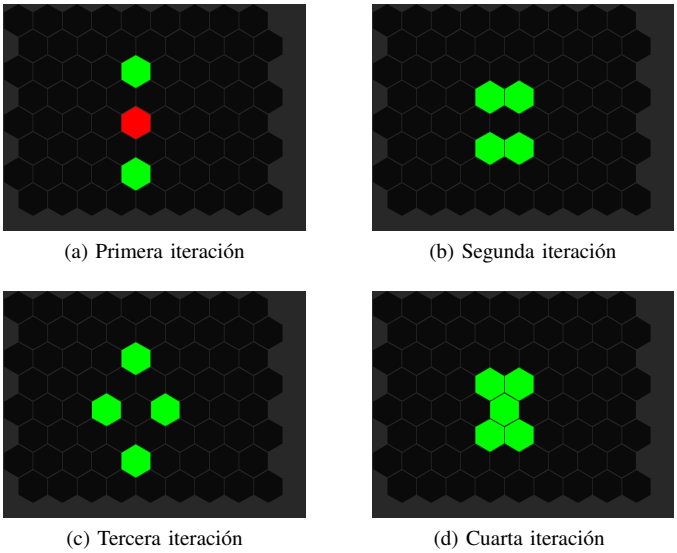


(a) Primera iteración (b) Segunda iteración

(c) Tercera iteración (d) Cuarta iteración

Fig. 14. (B2S2) Planeador

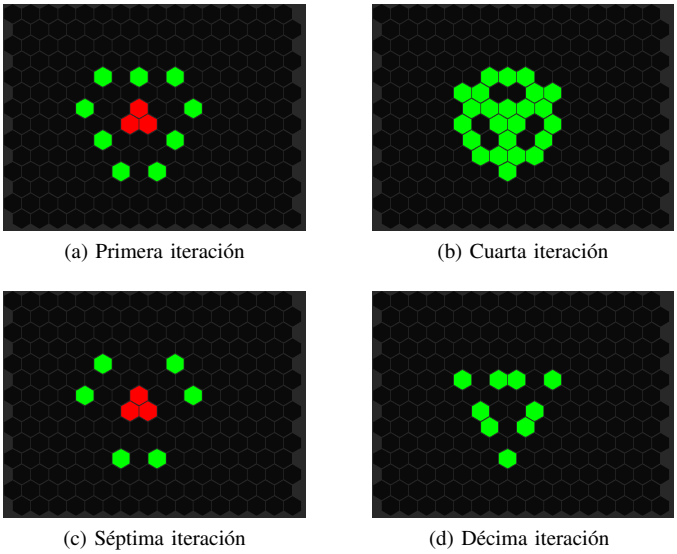
- **B2/S34** Esta regla es una de las más similares al Juego de la Vida original que hemos podido observar. Para una configuración inicial aleatoria con un 40% de células vivas, tanto en la malla hexagonal como de Voronoi, el resultado es un decaimiento hacia un estado en el que abundan los osciladores. Algunos de estos osciladores han sido documentados en un sitio web dedicado específicamente a esta regla [23]. Hemos podido observar una gran cantidad de osciladores, pero ningún planeador ni vida estática. En las Fig. 15, 16, 17, 18 mostramos algunos de los osciladores encontrados. En las Fig. 19, mostramos los estados obtenidos de la evolución de estados aleatorios en mallas hexagonales y de Voronoi.



(a) Primera iteración (b) Segunda iteración

(c) Tercera iteración (d) Cuarta iteración

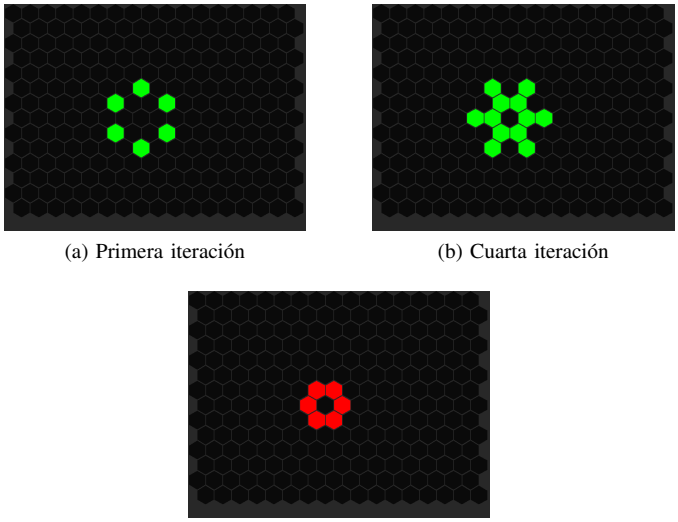
Fig. 15. (B2/S34) Oscilador



(a) Primera iteración (b) Cuarta iteración

(c) Séptima iteración (d) Décima iteración

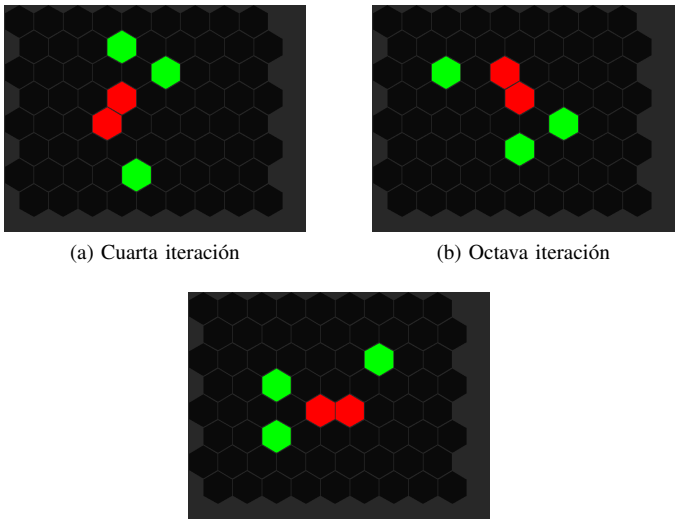
Fig. 16. (B2S34) Oscilador de periodo 12



(a) Primera iteración (b) Cuarta iteración

(c) Séptima iteración

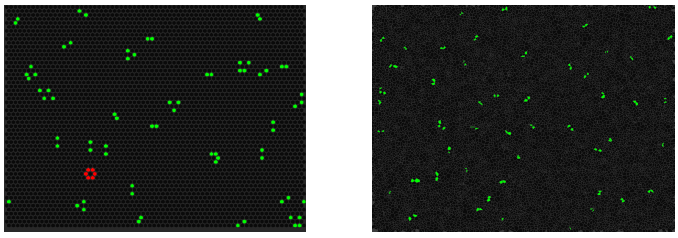
Fig. 17. (B2/S34) Oscilador de periodo 3



(a) Cuarta iteración (b) Octava iteración

(c) Doceava iteración

Fig. 18. (B2/S34) Otro oscilador de periodo 12



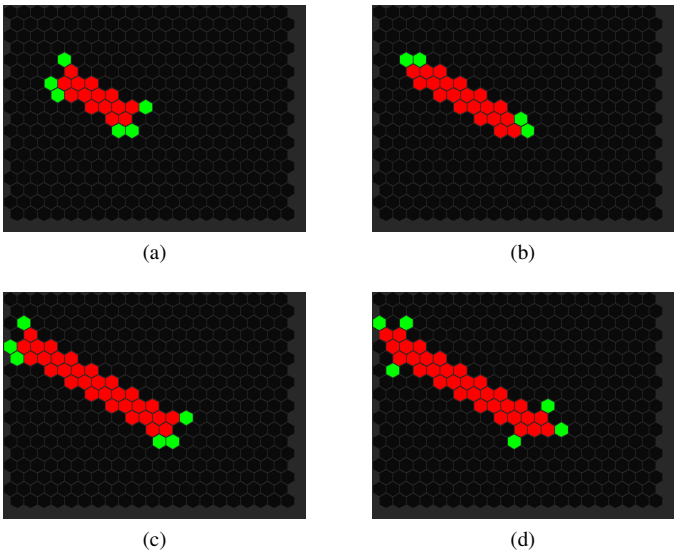
(a) (b)

Fig. 19. (B2/S34) Comparación de la malla hexagonal y el diagrama de Voronoi

- **B2/S345** Esta regla surge como una pequeña modificación de la regla B2/S34. En este caso, las células con cinco vecinos también pueden sobrevivir. En las Fig. 22, 23, 24, se muestran osciladores encontrados bajo esta regla. En las Fig. 20, 25 se muestran patrones de especial interés. Se trata de locomotoras o *puffers*: estructuras que se propagan por la malla dejando tras de sí un rasto de vida estática.

Por otro lado, en la Fig 21 mostramos un patrón de células vivas estable. Sin embargo, hemos de recurrir a la periodicidad de la malla para obtenerlo.

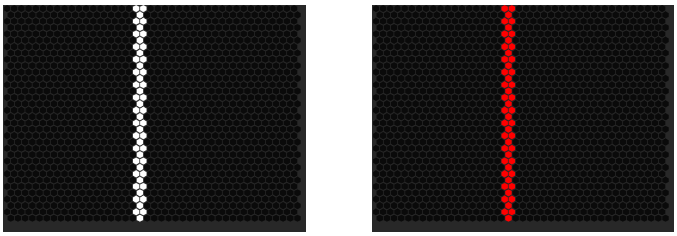
En cuanto al comportamiento de la malla de Voronoi, en la Fig. 26 podemos observar que el autómata tiende a crecer descontroladamente en la malla hexagonal, mientras que da lugar a una configuración de abundantes osciladores en el diagrama de Voronoi.



(a) (b)

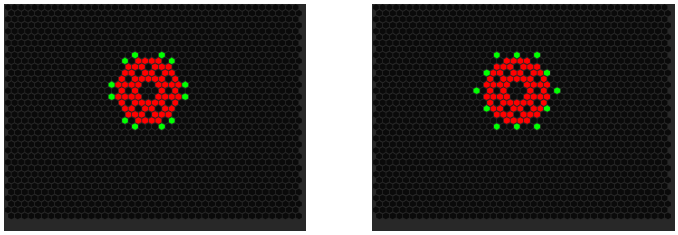
(c) (d)

Fig. 20. (B2/S345) Locomotora: patrón que se propaga por la malla dejando un rastro de célula vivas.



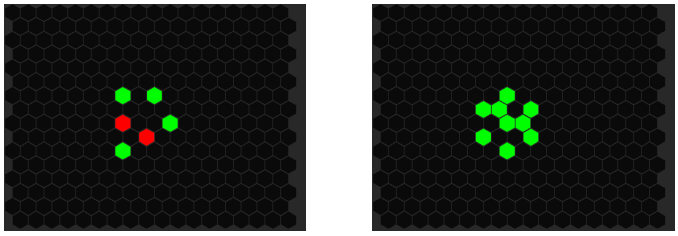
(a) (b)

Fig. 21. (B2/S345) Vida estática

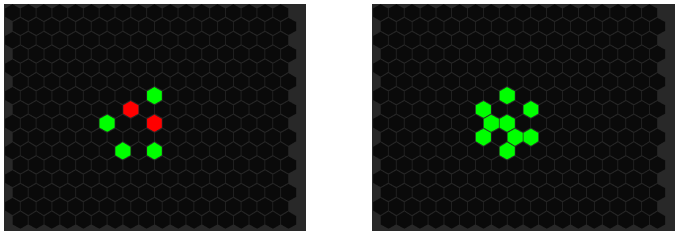


(a) (b)

Fig. 22. (B2/S345) Pseudo-vida estática

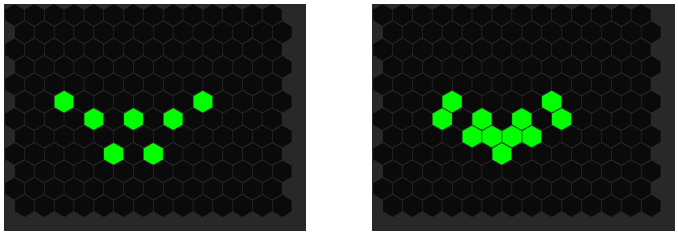


(a) (b)

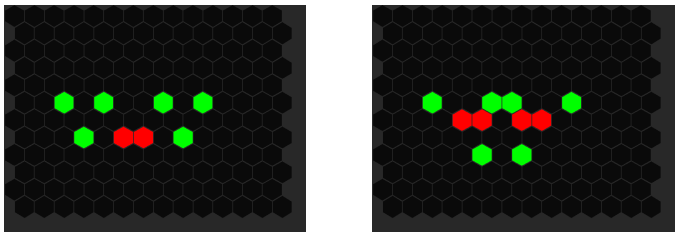


(c) (d)

Fig. 23. (B2/S345) Oscilador de periodo 4

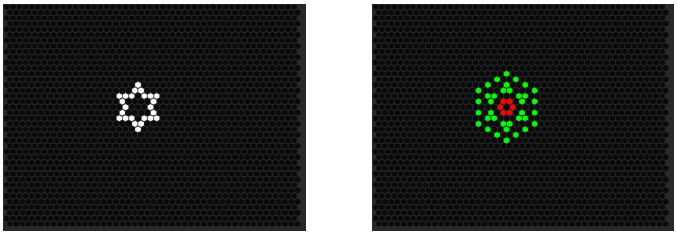


(a) Primera iteración (b) Segunda iteración

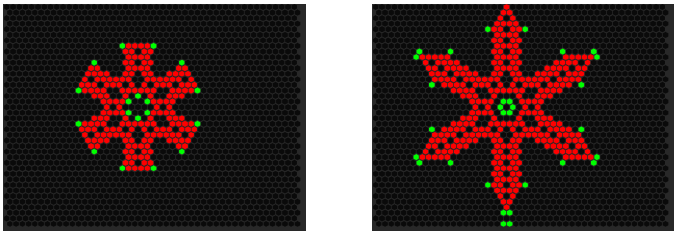


(c) Tercera iteración (d) Quinta iteración

Fig. 24. (B2/S345) Oscilador de periodo 5

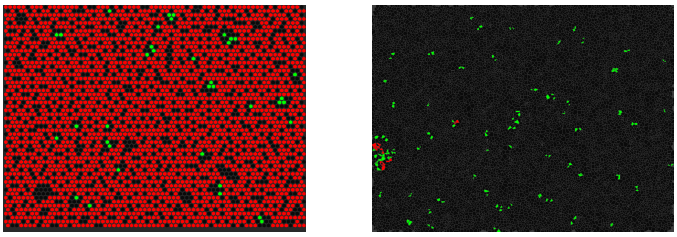


(a) (b)



(c) (d)

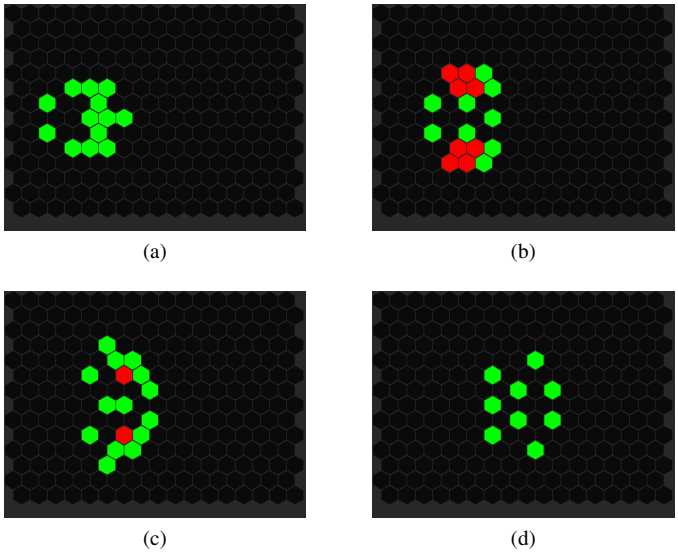
Fig. 25. (B2/S345) Generador de locomotoras.



(a) (b)

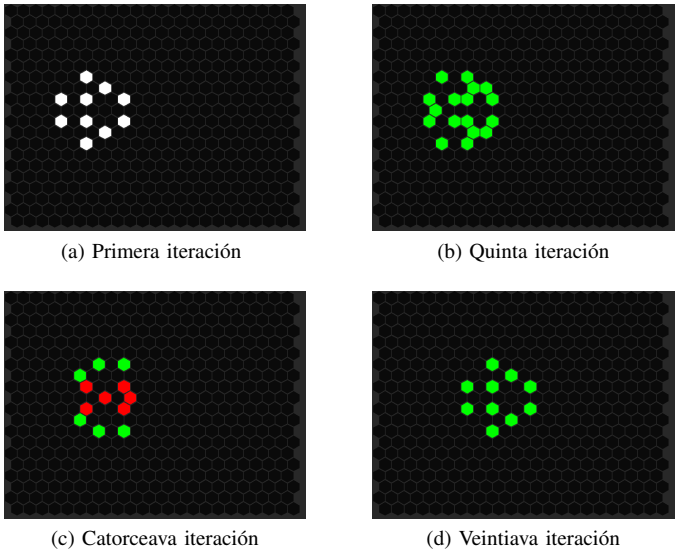
Fig. 26. (B2/S345) Comparación de la malla hexagonal y el diagrama de Voronoi

- **B245/S3 y B2/S3.** Consultando en el artículo “Cellular Automata in Triangular, Pentagonal and Hexagonal Tessellations” [7], hemos podido comprobar que se han documentado otros planeadores para mallas hexagonales. Los mostramos en las Fig. 27, 28.



(a) (b) (c) (d)

Fig. 27. (B245/S3) Planeador



(a) Primera iteración (b) Quinta iteración (c) Catorceava iteración (d) Veintiava iteración

Fig. 28. (B2/S3) Planeador

V. Conclusiones

Seguidamente, se concluye el reporte con algunas conclusiones y debates de posibles líneas de trabajo futuro.

A. Conclusiones

Uno de los autómatas celulares más destacables es el Juego de la Vida de John Conway. En este trabajo se desarrolla una herramienta de software con una finalidad doble:

- Explorar autómatas celulares similares al Juego de la Vida en mallas no cuadradas.
- Crear una aplicación con fines didácticos.

A lo largo de este documento, se han descrito los componentes del programa y las diferentes fases de nuestro proyecto. Posteriormente hemos mostrado el uso que le hemos dado a nuestro programa para aprender sobre el comportamiento que tienen los autómatas en las mallas hexagonales y de diagrama de Voronoi.

Los autómatas celulares sobre malla hexagonal han mostrado osciladores, planeadores y vida estática, así como locomotoras. Sobre las mallas de diagrama de Voronoi, en cambio, sólo hemos encontrado vida estática y osciladores. No encontramos planeadores ni locomotoras debido a que la malla no es periódica en el espacio.

B. Trabajo futuro

En base a los resultados obtenidos en este estudio, existen muchas áreas de investigación futura muy interesantes. En este apartado discutiremos las líneas más importantes.

- **Implementar otras formas de generación de mallas:** En este trabajo, hemos visto como funcionan los autómatas celulares en mallas cuadradas, hexagonales o haciendo uso de diagramas de Voronoi, pero, sería interesante probar otras formas de generación de mallas como puede ser *Penrose rhombii*, mallas triangulares o la teselación pentagonal de El Cairo.
  El generar otras mallas podría ser útil a la hora de realizar nuevas aplicaciones en el mundo real.
- **Autómatas celulares en 3D:** En general, la mayoría de estudios sobre autómatas celulares que se pueden observar son en 2 dimensiones, pero para poder resolver problemas prácticos o tener una mayor comprensión de algunos fenómenos que se están estudiando, es interesante poder adaptar nuestro programa a autómatas celulares tridimensionales.
  El problema es que deberíamos buscar otra librería gráfica que no sea Pygame, a virtud de que no tiene soporte nativo para gráficos en 3D (Pygame es un wrapper de la biblioteca Simple DirectMedia Layer, que es una API que trabaja en 2D). Por lo cual, si queremos realizar esto, deberemos hacer uso de librerías como PyOpenGL.
- **Aplicación a problemas prácticos:** En este reporte hemos realizado experimentos muy simples, pero sería conveniente poder aplicar este software a problemas prácticos, ya que podríamos crear nuevas ideas y soluciones que pueden tener efectos positivos en diferentes campos de estudio como la biología o el diseño de materiales.
  Un ejemplo de cómo el campo del diseño de materiales podría beneficiarse es en el estudio del crecimiento de los copos de nieve utilizando autómatas celulares hexagonales [24]. Comprender este proceso básico puede ayudarnos a desarrollar materiales cristalinos más avanzados.
  Otro ejemplo de aplicación a problemas prácticos sería el poder predecir la propagación de un frente de fuego en ambientes forestales homogéneos y no homogéneos, haciendo uso de autómatas celulares que representan áreas hexagonales del bosque [8].
- **Optimización del programa:** Aunque nuestro programa funciona bien, se pueden aplicar métodos de optimización de código para mejorar el rendimiento y la eficiencia de

nuestro software. Esto se hace con la finalidad de ahorrar tiempo y recursos computacionales, contribuyendo a la protección del desarrollo sostenible.

APÉNDICE A
MANUAL DE USUARIO

A continuación, se procede a describir tanto los sistemas compatibles con el programa, así como las dependencias necesarias para poder ejecutar el programa realizado.

A. Sistemas recomendables

El proyecto ha sido desarrollado haciendo uso del lenguaje de programación Python. Y aunque este lenguaje es compatible con la mayoría de sistemas de escritorio personales, se ha hecho uso de funciones donde manipulamos la estructura de directorios. Esto significa que no tenemos la certeza de que en todos los sistemas pueda funcionar.

El desarrollo de este software se ha realizado utilizando Windows 10, por lo que se recomienda hacer uso de este sistema.

Si tiene cualquier problema a la hora de poder ejecutar el programa, puede abrir una propuesta accediendo al siguiente enlace: https://github.com/kennyfh/PyCA/issues

B. Dependencias

Además de tener instalado el lenguaje de programación Python con una versión igual o superior a la 3.7, debemos tener los siguientes módulos:

- **Pygame**
- **Tkinter**
- **Scipy**
- **Imageio**

Si no quieres estar instalando las dependencias de forma manual, podemos instalar todas las dependencias usando el fichero *requirements.txt* dentro del repositorio. Para ello puedes usar el siguiente comando en una terminal que esté en el directorio raíz del repositorio:

```
pip install -r requirements.txt
```

Si tiene cualquier duda, puede consultar la guía de instalación que tenemos en nuestra wiki entrando al siguiente enlace: https://github.com/kennyfh/PyCA/wiki/Installation

C. Ejecución del software

El código del programa puede ser descargado a través del repositorio hospedado en https://github.com/kennyfh/PyCA.

Una vez instaladas todas las dependencias anteriores y el proyecto descargado, el programa puede ser ejecutado haciendo uso del siguiente comando desde la carpeta raíz del proyecto:

```
python scripts/main.py
```

Sin embargo, si no quieres descargarte el repositorio, otra opción es descargarte el ejecutable desde la sección *release* de nuestro repositorio.

Para mayor información consultar nuestra wiki desde el siguiente enlace: https://github.com/kennyfh/PyCA/wiki.

REFERENCIAS

[1] Lifewiki, the wiki for conway's game of life, 2022. URL https://conwaylife.com/wiki/.

[2] Alejandro Salcido. *Emerging Applications of Cellular Automata*. INTECH, 05 2013. ISBN 978-953-51-1101-6.

[3] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 1 edition, 2002. ISBN 1579550088; 9781579550080.

[4] Vural Erdogan. 2d hexagonal cellular automata: The complexity of the forms, 2018. URL https://arxiv.org/abs/1811.12387.

[5] Christopher G. Langton. Artificial Life : *the Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems*. Addison-Wesley, Los Alamos, New Mexico, 1989.

[6] Calvin's Hobbies. Implement the Game of Life on Anything but a Regular Grid, June 2020. URL https://codegolf.stackexchange.com/q/35827.

[7] Carter Bays. *Cellular Automata in Triangular, Pentagonal and Hexagonal Tessellations*, pages 892–900. Springer New York, New York, NY, 2009. ISBN 978-0-387-30440-3. doi: 10.1007/978-0-387-30440-3_58. URL https://doi.org/10.1007/978-0-387-30440-3_58.

[8] L. Hernández Encinas, S. Hoya White, A. Martín del Rey, and G. Rodríguez Sánchez. Modelling forest fire spread using hexagonal cellular automata. *Applied Mathematical Modelling*, 31(6):1213–1227, 2007. ISSN 0307-904X. doi: https://doi.org/10.1016/j.apm.2006.04.001. URL https://www.sciencedirect.com/science/article/pii/S0307904X06000916.

[9] Andrew Adamatzky, Andrew Wuensche, and Benjamin De Lacy Costello. Glider-based computing in reaction-diffusion hexagonal cellular automata. *Chaos, Solitons & Fractals*, 27(2):287–295, 2006. ISSN 0960-0779. doi: https://doi.org/10.1016/j.chaos.2005.03.048. URL https://www.sciencedirect.com/science/article/pii/S0960077905003541.

[10] D. D'Ambrosio, S. Di Gregorio, and G. Iovine. Simulating debris flows through a hexagonal cellular automata model: SCIDDICA S_{3-hex}. *Natural Hazards and Earth System Sciences*, 3(6):545–559, December 2003. ISSN 1561-8633. doi: 10.5194/nhess-3-545-2003. URL https://nhess.copernicus.org/articles/3/545/2003/.

[11] Irfan Siap, Hasan Akin, and Selman Uğuz. Structure and reversibility of 2D hexagonal cellular automata. *Computers & Mathematics with Applications*, 62(11):4161–4169, December 2011. ISSN 08981221. doi: 10.1016/j.camwa.2011.09.066. URL https://linkinghub.elsevier.com/retrieve/pii/S0898122111008479.

[12] Hisamoto Hiyoshi, Yukari Tanioka, Toshiki Hamamoto, Kazuya Matsumoto, and Kaito Chiba. Pedestrian movement model based on voronoi cellular automata. *Transportation Research Procedia*, 2:336–343, 2014. ISSN 2352-1465. doi: https://doi.org/10.1016/j.trpro.2014.09.027. URL https://www.sciencedirect.com/science/article/

pii/S2352146514000635. The Conference on Pedestrian and Evacuation Dynamics 2014 (PED 2014), 22-24 October 2014, Delft, The Netherlands.

[13] Wenzhong Shi and Matthew Yick Cheung Pang. Development of voronoi-based cellular automata -an integrated dynamic model for geographical information systems. *International Journal of Geographical Information Science*, 14(5):455–474, 2000. doi: 10.1080/13658810050057597. URL https://doi.org/10.1080/13658810050057597.

[14] Uri Wilensky. Netlogo, 1999. URL http://ccl.northwestern.edu/netlogo/.

[15] Golly Game of Life Home Page, 2018. URL https://golly.sourceforge.net/.

[16] Alexander S. Gillis. What is Object-Oriented Programming (OOP)?, 2020. URL https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP.

[17] Paul Callahan. Experiments with a somewhat "Life-like" hexagonal CA, December 1997. URL http://www.radicaleye.com/lifepage/hexrule.txt.

[18] Andrew Wuensche. Glider dynamics in 3-value hexagonal cellular automata: The beehive rule. *Int. J. Unconv. Comput.*, 1:375–398, 01 2005.

[19] Alex Kazakov. Hexalattice, 2020. URL https://github.com/alexkaz2/hexalattice.

[20] imageio/imageio: v2.24.0, January 2023. URL https://zenodo.org/record/1488561.

[21] PyInstaller Development Team. Pyinstaller, 2010. URL https://www.pyinstaller.org/.

[22] Conway's Game of Life - LifeWiki, 2008. URL https://conwaylife.com/wiki/Conway's_Game_of_Life.

[23] Hexlife - ./home/davidsiaw. URL https://davidsiaw.github.io/blog/2014/11/21/hexlife/.

[24] Jessica C. Li. On the modeling of snowflake growth using hexagonal automata. 2015.