Instituto Politécnico Nacional
Escuela Superior de Cómputo
Materia: GENETIC ALGORITHMS
Lab Session 9. CA
Alumno: Torres Abonce Luis Miguel
Profesor: Jorge Luis Rosas Trigueros
Fecha de Realización: 14 Noviembre 2024
Fecha de Entrega: 19 Noviembre 2024

Marco Teórico.

Los Autómatas Celulares (CA) son modelos computacionales y matemáticos que permiten simular sistemas dinámicos complejos a través de reglas simples. Fueron introducidos por el matemático John von Neumann en los años 40, pero popularizados por John Conway con su famoso "Juego de la Vida" en 1970.

Un autómata celular se basa en una rejilla discreta compuesta por celdas, donde cada celda puede adoptar un estado específico. Estos estados evolucionan en función de reglas locales que dependen del estado de las celdas vecinas. Este enfoque permite modelar fenómenos físicos, biológicos y computacionales de forma eficiente.

Características principales

1. Espacio discreto: Las celdas están dispuestas en una cuadrícula unidimensional, bidimensional o multidimensional.
2. Estados finitos: Cada celda puede tener un número finito de estados (e.g., "viva" o "muerta").
3. Reglas locales: La evolución de una celda depende de su estado actual y el de sus celdas vecinas.
4. Iteraciones: Los cambios de estado ocurren en pasos discretos de tiempo, llamados generaciones.

Aplicaciones

Los autómatas celulares se utilizan en múltiples disciplinas, incluyendo:

- Modelado de crecimiento de organismos biológicos.
- Simulación de sistemas físicos como fluidos o propagación de incendios.
- Teoría de la computación (e.g., demostración de la universalidad computacional).
- Generación de patrones visuales y arte generativo.

El Juego de la Vida (Game of Life) es un autómata celular bidimensional diseñado por John Conway. Es un modelo matemático simple que puede producir comportamientos increíblemente complejos a partir de reglas básicas.

Reglas del Juego de la Vida

1. Supervivencia: Una celda viva permanece viva si tiene exactamente 2 o 3 vecinos vivos.
2. Nacimiento: Una celda muerta se convierte en una celda viva si tiene exactamente 3 vecinos vivos.
3. Muerte por soledad o sobrepoblación: Las celdas vivas con menos de 2 vecinos mueren por soledad; aquellas con más de 3 vecinos mueren por sobrepoblación.

El Juego de la Vida es un autómata universal, lo que significa que puede simular cualquier máquina de Turing y realizar cualquier cálculo computable.

Fenómenos clave

- Osciladores: Patrones que repiten su configuración en ciclos (e.g., "Blinker").
- Naves espaciales: Patrones que se mueven a través de la cuadrícula (e.g., "Glider").

- Estabilidad: Patrones que permanecen estáticos (e.g., "Bloque").
- Emergencia: Patrones complejos que surgen de interacciones simples.

Golly es un software especializado en la simulación y exploración de autómatas celulares. Fue diseñado para ser altamente eficiente y soportar cuadrículas de tamaño prácticamente ilimitado. Golly es especialmente conocido por su implementación del Juego de la Vida, aunque también permite simular otros autómatas celulares.

Características principales de Golly

1. Algoritmos avanzados: Utiliza el algoritmo Hashlife, que permite simular generaciones rápidamente incluso en configuraciones enormes.
2. Compatibilidad: Soporta una variedad de reglas, incluidas las personalizadas por el usuario.
3. Interfaz gráfica: Permite la visualización de patrones en tiempo real.
4. Compatibilidad con formatos: Puede importar y exportar patrones en formatos estándar, como .rle (run-length encoded).
5. Librerías integradas: Incluye patrones famosos, como osciladores, naves espaciales y "máquinas de computación".

La simulación de autómatas celulares en herramientas como Golly permite estudiar cómo reglas simples pueden generar comportamientos altamente complejos. En el caso del Juego de la Vida, los usuarios pueden observar fenómenos como el crecimiento exponencial de patrones, la creación de estructuras autorreplicantes y la computación de problemas lógicos. Golly se ha convertido en un estándar para la experimentación con autómatas celulares gracias a su velocidad, flexibilidad y facilidad de uso. Su capacidad para manejar configuraciones extremadamente grandes lo hace ideal para explorar las propiedades teóricas y aplicadas de estos sistemas.

Material y Equipo

- **Hardware:** Computadora con capacidad para ejecutar Python.
- **Software:**
  - Python 3.x
    Google Colab.

Desarrollo de la practica

- Los osciladores cambian de forma y después de un tiempo vuelven a su configuración original en ciclos.
- Los still lifes permanecen inmóviles.
- El glider moviéndose a través de la cuadrícula, repitiendo la misma figura.

1. *Con los últimos 6 dígitos de mi número de estudiante "6,3,0,7,3,8", se encontraron las siguientes, still life, glider y osciladores:*

Figura 1. Still life, esta se obtuvo con la regla B3/S0

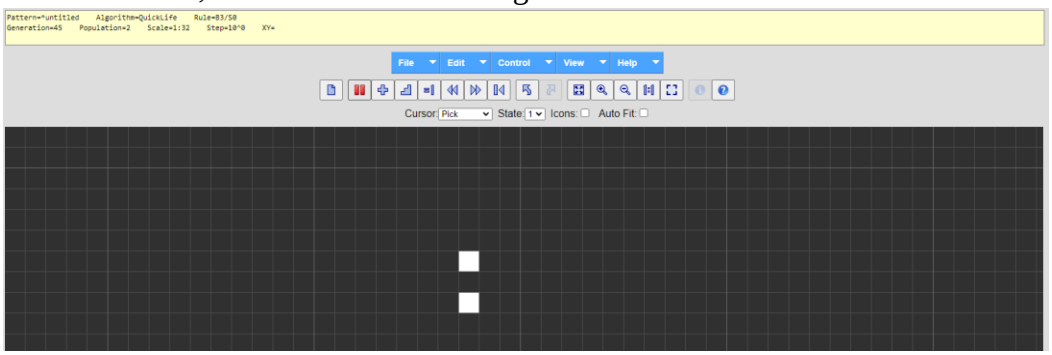

Figura 1

Figura 2. Oscilador, Esta se obtuvo con la regla B0/S3

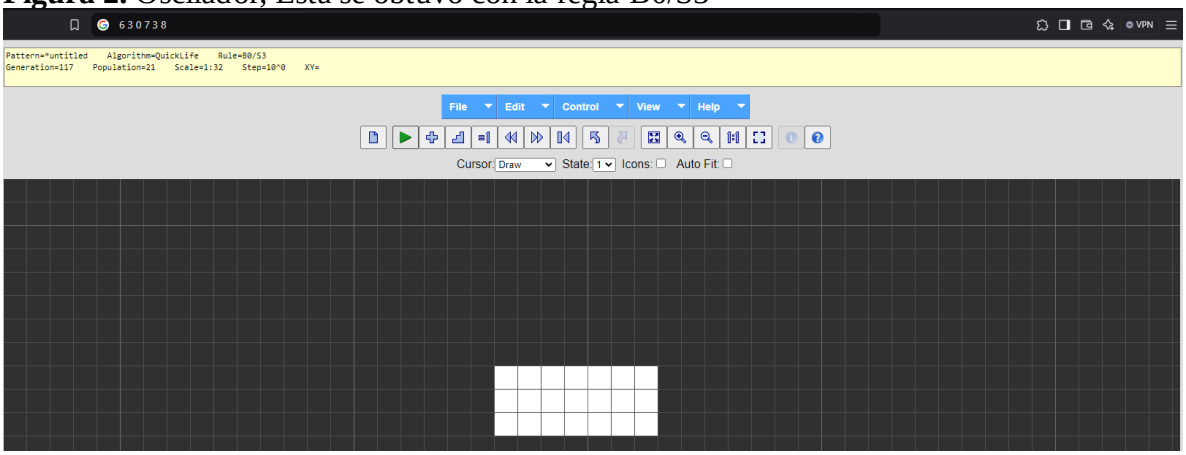

Figura 2.

Figura 3. Glider, este se obtuvo con la regla b36/23

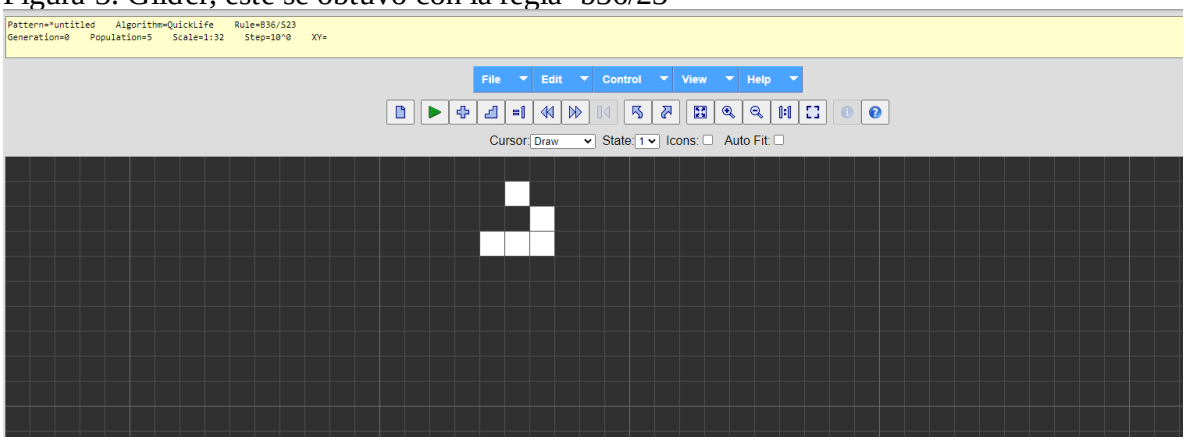

Figura 3.

2. *Encontrar con reglas un oscilador, Still life y Glider en un autómata celular unidimensional (1D), con al menos 3 células en su estado inicial*

Figura 4. Still life se encontró con la regla w0.

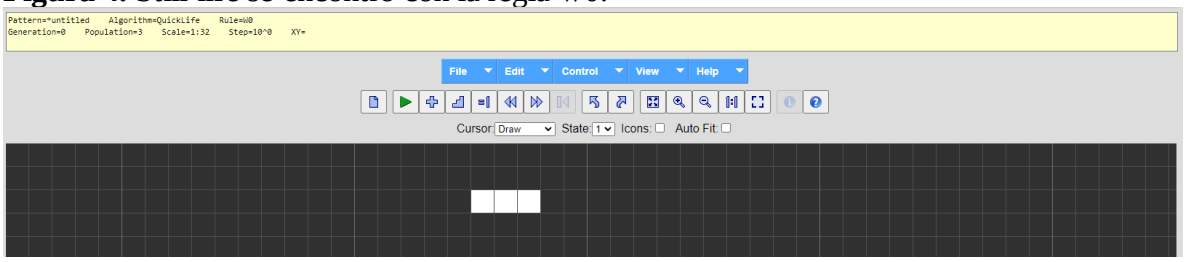

Figura 4.

Figura 5. Oscilador

Figura 6. Glider. Se encontró un glidder con la regla w4.

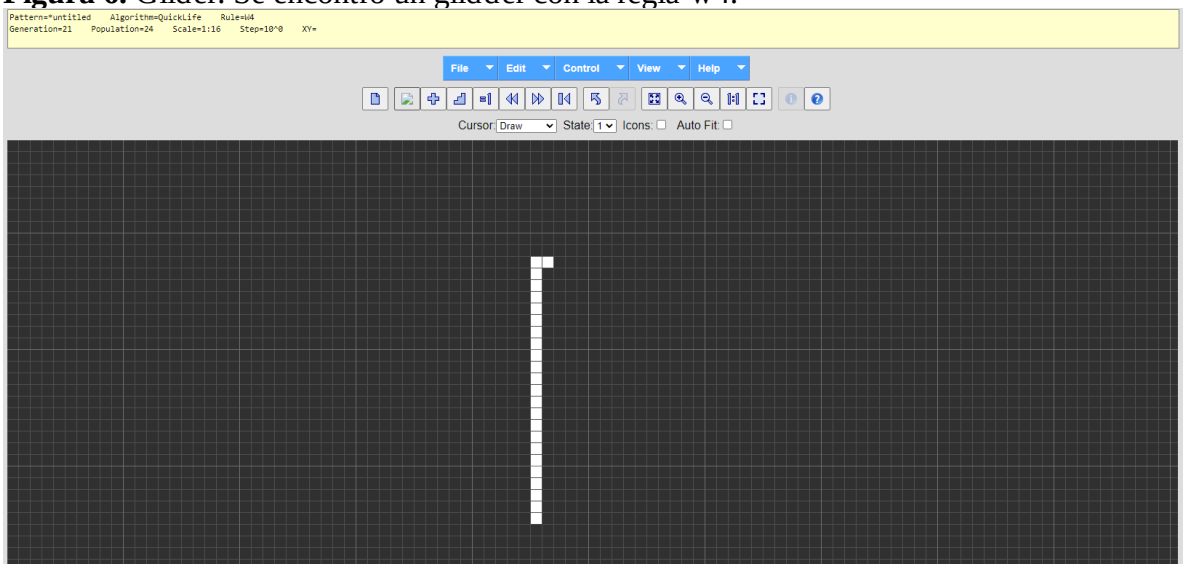

Figura 6.

3. *Ahora realizamos código para una CA 1D de segundo orden con dos estados donde se pueda definir la regla a utilizar.*

Se realizo de la siguiente forma:

1. Autómata celular unidimensional
   - El autómata utiliza una sola dimensión de celdas, representadas como una lista de valores. Cada celda puede estar en uno de dos estados: activo (1) o inactivo (0).
   - La evolución se realiza actualizando los estados de las celdas a lo largo de varias generaciones, siguiendo una regla específica para calcular el siguiente estado.
2. Segundo orden
   - En este tipo de autómata, el estado de una celda en la siguiente generación no solo depende de su vecindad actual, sino también de su estado en la generación anterior.
   - El código cumple esto al mantener dos listas: una para el estado actual y otra para el estado previo. Antes de actualizar el siguiente estado, verifica si la celda estuvo activa en el paso anterior. Si fue así, automáticamente se desactiva en el siguiente paso, independientemente de su vecindad.

3. Dos estados
   - Las celdas tienen exactamente dos estados posibles: activo (1) e inactivo (0).
   - El comportamiento del autómata asegura que todas las transiciones y reglas solo producen estos dos estados, lo que cumple con la definición de un sistema binario.
4. Regla definida
   - La evolución del autómata está gobernada por una regla definida por un número entre 0 y 255. Este número se convierte a una representación binaria de 8 bits, que define cómo responderá cada patrón de vecindad (por ejemplo, combinaciones de los estados de una celda y sus vecinas izquierda y derecha).
   - La regla personalizada garantiza flexibilidad, permitiendo que el usuario defina el comportamiento deseado para el autómata.
5. Condiciones de frontera
   - El autómata utiliza condiciones de frontera circulares, también llamadas envolventes. Esto significa que la primera celda considera como vecina a la última celda y viceversa.
   - Este enfoque asegura que las celdas en los extremos no queden fuera del cálculo, cumpliendo con las condiciones comunes para autómatas celulares.
6. Evolución temporal
   - El autómata evoluciona durante un número definido de pasos (generaciones). En cada paso, se actualizan los estados actuales considerando tanto los estados previos como las reglas definidas.
   - Se mantiene un historial de todos los estados generados, lo que permite observar la evolución del sistema a lo largo del tiempo.
7. Visualización del resultado
   - El código produce una representación visual de la evolución del autómata, donde los estados activos (1) se muestran como bloques sólidos y los inactivos (0) como espacios vacíos. Esto permite analizar de manera intuitiva el comportamiento dinámico del sistema.

Figura 7. Esta imagen muestra la evolución de un autómata celular unidimensional de segundo orden durante varios pasos. La estructura indica cómo las celdas activas (1) y las inactivas (0) interactúan y evolucionan bajo las reglas definidas.

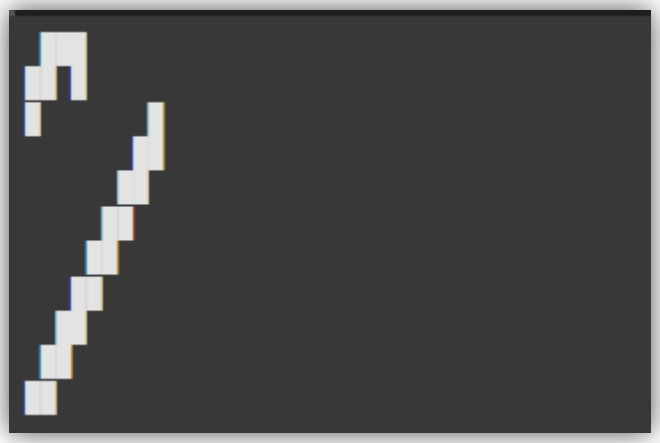

Figura 7.

Conclusiones

El autómata celular desarrollado demuestra cómo los sistemas de segundo orden con dos estados son capaces de generar dinámicas complejas y patrones específicos. Este tipo de sistema es útil para modelar fenómenos donde el estado presente y la memoria del sistema (estado previo) tienen una influencia significativa en la evolución. La práctica resalta la importancia de definir correctamente las reglas y condiciones iniciales para obtener el comportamiento deseado en este tipo de sistemas.

GOLLY

- **Autor:** John Horton Conway
- **Fuente:** https://golly.sourceforge.io/

Autómatas Celulares

- **Autor:** Universidad Sevilla
- **Fuente:** https://www.cs.us.es/~fsancho/Blog/posts/Automatas_Celulares.md.html

Anexo:

Código CA:

```
def second_order_ca(rule_number, initial_state, steps):
    # Convertir el número de regla a representación binaria de 8 bits
    rule_bin = f"{rule_number:08b}"
    rule = {f"{i:03b}": int(bit) for i, bit in enumerate(reversed(rule bin))}

    # Inicializar los estados previos y actuales
    prev_state = [0] * len(initial_state)
    current_state = initial_state.copy()

    # Guardar la historia de estados
    history = [current_state.copy()]

    for _ in range(steps):
        next_state = []
        for i in range(len(current_state)):
            # Obtener vecinos con envoltura (boundary conditions)
            left = current_state[(i - 1) % len(current_state)]
            center = current_state[i]
            right = current_state[(i + 1) % len(current_state)]

            # Incluir el estado previo de la célula
            prev = prev state[i]

            # Formar el patrón de vecinos
            neighborhood = f"{left}{center}{right}"

            # Aplicar la regla considerando el estado previo
            if prev == 1:
```

```
                # Si la célula estaba activa en el paso anterior, se desactiva
                next_cell = 0
            else:
                next_cell = rule[neighborhood]
            next_state.append(next_cell)

        # Actualizar estados para el siguiente paso
        prev_state = current_state.copy()
        current_state = next_state.copy()
        history.append(current state.copy())

    return history

# Ejemplo de uso:
# Definir la regla (por ejemplo, regla 110), el estado inicial y el número de pasos
rule number = 110
initial state = [0, 1, 1, 1, 0, 0, 0, 0, 0]
steps = 10

result = second_order_ca(rule_number, initial_state, steps)

# Imprimir el resultado
for state in result:
    print(''.join(['' if cell else ' ' for cell in state]))
```