Trabajo Final Simulación

Juego de la vida de Conway y Gardner.

Uno de los ejemplos más difundidos de autómatas celulares es el denominado Juego de la Vida (Game of Life), publicado en la revista matemática *Mathematical Games* por Martin Gardner y desarrollado por John Conway en 1970. El mismo consiste en una disposición bidimensional de casilleros que conforman una cuadrícula. Considerado cada casillero como una célula, esta puede tener dos estados: viva o muerta. El vecindario de cada célula está conformado por las cuatro células adyacentes lateralmente más las cuatro células adyacentes de manera diagonal. Las reglas de evolución son las siguientes:

- Si una célula viva tiene dos o tres vecinas vivas entonces sobrevive. Caso contrario muere, ya sea por aislamiento o por superpoblación.
- Si el vecindario de una célula muerta consta de tres células vivas entonces se produce un nacimiento. Caso contrario, la célula permanece muerta.

Una posible trayectoria se puede observar en la **Figura 1**.

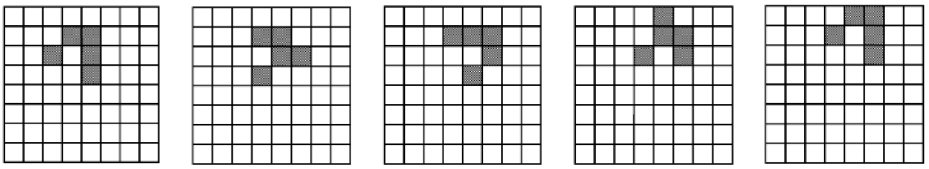

Figura 1: Una posible trayectoria del Juego de la Vida.

Consecuentemente, cada célula puede pensarse como un modelo de tiempo discreto, el cual consta de ocho entradas (las salidas de las células pertenecientes a su vecindario), una variable de estado que indica si está viva o muerta (podríamos suponer que es una variable numérica que vale 0 si está muerta y 1 si está viva), y una salida, que coincide con el estado.

Un programa que permita simular un modelo de este tipo, bajo un enfoque de tiempo discreto deberá simular cada célula a cada instante

de tiempo para poder determinar su estado futuro. Esto es, deberá evaluar la función de transición de cada célula en cada iteración. Si suponemos que el sistema tiene dimensiones relativamente grandes, sin duda este tipo de simulación tendrá enormes desventajas en lo relativo al tiempo de cálculo.

Existe sin embargo una simplificación al problema de la simulación de este ejemplo. Si en el vecindario de una célula no se produjo ningún cambio en la n-ésima iteración, sin dudas en dicha célula no se producirá ningún cambio en la siguiente iteración (en tiempo n+1) y por consiguiente, cuando no se produzcan cambios en el vecindario no será necesario evaluar qué ocurrirá en la célula en la siguiente iteración.

Especificación

Para la especificación, el componente de una célula fue planteado según se remarca en la consigna de introducción al problema. Además se puede comprobar la representación como se muestra en la **Figura 2**.

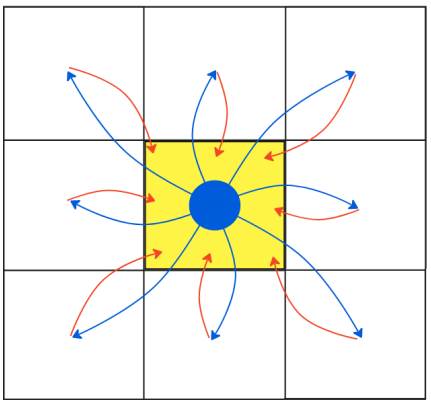

Figura 2: Representación utilizada como base del modelo de una célula y su vecindario.

En donde se puede observar que cada célula está representada por un puerto de salida (marcado en color azul), y ocho puertos de entrada (marcados en color rojo) cuyas conexiones provienen del vecindario asociado a dicha célula. En el modelo también se contemplan los casos en donde la cantidad de vecinos no es de ocho (casos borde o esquinas del tablero de células).

A continuación se muestra la especificación del problema siguiendo el formalismo DEVS:

GameOfLife = <X, Y, S, δ_{int}, δ_{ext}, λ, ta>

X = {(v, p) | v ∈ {1, 0}, p ∈ IPorts}

Y = {(v, p) | v ∈ {1, 0}, p ∈ OPorts}

S = {v | v ∈ {1, 0}} x {1, 0}$^+$ x **Z** x **Z** x $\mathbf{R}_0^+$

λ(s)= λ((isAlive, neighbors, neighborsUpdated,
maxNeighbors, σ))= (isAlive,0)

ta(s)= ta((isAlive, neighbors, neighborsUpdated,
maxNeighbors, σ))= σ

δ_{int}(s)= δ_{int}((isAlive, neighbors, neighborsUpdated,
maxNeighbors, σ))= (isAlive, neighbors, neighborsUpdated,
maxNeighbors, TRANSITION_TIME)

δ_{ext}(s, e, x)= δ_{ext}((isAlive, neighbors, neighborsUpdated,
maxNeighbors, σ), e, (v, p)) = updateState(isAlive,
neighbors, neighborsUpdated, maxNeighbors, σ, e, v, p)

```
updateState(isAlive, neighbors, neighborsUpdated,
maxNeighbors, σ, e, v, p)
{
      neighbors[p] = v
      neighborsUpdated++
      if (neighborsUpdated == maxNeighbors)
      {
            countAndUpdate(isAlive, neighbors,
            neighborsUpdated)
      }
      σ = σ - e

      return (isAlive, neighbors, neighborsUpdated,
            maxNeighbors, σ)
}
```

```
countAndUpdate(isAlive, neighbors, neighborsUpdated)
{
      livenessCounter = 0
      for (i = 0, i < NEIGHBORS_AMOUNT, i++)
      {
            if (neighbors[i])
            {
                  livenessCounter++
            }
      }

      if (isAlive && (livenessCounter < A ||
       livenessCounter > B))
      {
            isAlive = 0
      }
      else if (!isAlive && livenessCounter == BIRTH)
      {
            isAlive = 1
      }
      neighborsUpdated = 0
}
```

Antes de la explicación completa del funcionamiento del modelo, es necesario indicar que se tomó en cuenta la existencia de las siguientes constantes globales que se utilizan en algunas funciones:

- "`TRANSITION_TIME`": Tiempo de transición entre cada estado de la simulación. Es elegido por el usuario.
- "`NEIGHBORS_AMOUNT`": Cantidad total de vecinos que pueden tener todas las células. Siempre son ocho.
- "`A`" y "`B`": Intervalo de vivacidad, es decir, la cantidad mínima y máxima, respectivamente, de células vecinas que se requieren para que la actual se mantenga con vida. Son elegidas por el usuario (Aunque la consigna remarca que se deben ser 2 y 3).
- "`BIRTH`": Cantidad de células vecinas necesarias para que se genere un nacimiento de la actual en el próximo estado. Es elegido por el usuario (Aunque la consigna remarca que debe ser 3).

La especificación se pensó en términos de un estado inicial determinado para las células, ya que hay características particulares a considerar para algunas y para otras no, como por ejemplo, que no

todas van a tener ocho vecinos, y esto es algo que infiere en la generación del estado de cada célula.

A continuación se hace la descripción de cada parte de la especificación realizada:

- Conjunto de entrada ("`X`"): Es un conjunto de tuplas que contienen valores entre "0" y "1", pensados a ser los estados de las células vecinas de la actual (muertas o vivas, respectivamente); en donde cada vecino entra por un puerto diferente de entrada.
- Conjunto de salida ("`Y`"): Es un conjunto de tuplas que contienen valores entre "0" y "1", pensados para ser los estados de la célula actual (muerta o viva, respectivamente); en donde la salida se genera por el puerto de salida.
- Estado ("`S`"): Es una 5-upla pensada para ser denotada de la siguiente manera "`(isAlive, neighbors, neighborsUpdated, maxNeighbors, σ)`", en donde:
  - "`isAlive`": Es el estado actual de la célula (viva o muerta).
  - "`neighbors`": Es una lista o arreglo de los estados de las células vecinas. Esto significa que cada célula conoce el estado actual de sus vecinos.
  - "`neighborsUpdated`": Es una variable que funciona como contador, con la finalidad de dar permiso a que una célula actualice su estado en función de la cantidad de células vecinas que emitieron su salida.
  - "`maxNeighbors`": Es una constante que permite funciona junto a la variable anterior, ya que cada célula debe conocer cuántas entradas debe recibir para poder considerar actualizar su estado o no para la próxima iteración.
  - "σ": Variable que designa el tiempo de permanencia en cada estado.
- Función de salida ("λ"): Se retorna una tupla con el estado actual de la célula, por el puerto de salida 0; tal como se explica en el conjunto de salida.
- Función de tiempo de permanencia ("`ta`"): Se retorna "σ" ya que es la variable de tiempo de avance.

- Función de transición interna ("δ_{int}"): Se actualiza el tiempo de permanencia de cada estado con la constante "`TRANSITION_TIME`", ya que el problema se trata de un sistema de tiempos discretos.
- Función de transición externa ("δ_{ext}"): En general esta función modifica el estado de una célula una vez los vecinos hayan ejecutado todas las salidas (ya que, en el caso de generar la salida al mismo tiempo, estas se ejecutan de forma secuencial en el orden en el cuál fue creada cada célula). Básicamente cuando una célula vecina dispara una salida, se activa la función actual, que actualiza el listado de estados vecinos e incrementa la cantidad de vecinos actualizados ("`neighbors`" y "`neighborsUpdated`", respectivamente); a su vez, se actualiza el tiempo constantemente y se retorna el estado con los elementos modificados. Cuando todos los vecinos hayan disparado una salida, se ejecuta la función "`countAndUpdate`", que analiza el listado de vecinos para actuar de la forma que sea necesaria (si la célula se mantiene viva, se muere, nace, o se mantiene muerta).

Proyecto

Para la implementación del proyecto, se generó una carpeta "`game_of_life`", compuesto de otras carpetas y archivos, en donde se incluyen las instrucciones para la ejecución de las simulaciones que se quieran generar. A continuación se muestra el árbol de contención de archivos y se explica brevemente la funcionalidad de los mismos:

```
game_of_life
├── generate_model.py
├── generate_plot.py
├── input
├── life
│   ├── cell.cpp
│   ├── cell.h
│   └── constants.h
├── output
└── README.md
```

- "`generate_model.py`": Script en Python que permite crear un modelo de formato "`.pdm`" para correr una simulación en PowerDEVS.
    - Para la generación del modelo, el script toma un input de un determinado formato, en el que escribe los parámetros de la simulación y de cada una de las células. Además, se realizó un estudio sobre el formato de los modelos, para que cada uno de los que se genere pueda ser relativamente comprensible a la vista debido a la alta cantidad de conexiones entre cada una de las células.
- "`generate_plot.py`": Script en Python que permite crear un plot y una animación a partir del output de la simulación en PowerDEVS.
    - Para la generación de las animaciones el script captura los datos de salida obtenidos de la simulación, en el archivo "`pdevs.log`" ubicado en los directorios de la herramienta; y mediante el uso de la librería "Matplotlib" de Python, se genera la animación en formato "`.gif`". Además, el script incluye comentarios documentativos para poder modificar el código y realizar una observación de frame por frame de los estados de la simulación.
- "`input`": Carpeta que contiene los archivos de entrada para crear el modelo con un estado inicial y otros parámetros predeterminados.
- "`life`": Carpeta que contiene los códigos de implementación de la especificación mostrada, para los componentes que se utilizan en el modelo.
- "`output`": Carpeta que contiene las animaciones generadas por el script "`generate_plot.py`".
- "`README.md`": Instrucciones para la utilización del proyecto.

Patrones

A continuación se mostrarán algunos de los patrones con los que se experimentó, y se describirán las observaciones realizadas para cada uno de ellos.

1. Still life patterns

Estos patrones son configuraciones de celdas en el tablero que permanecen estáticas y no cambian en ninguna de las generaciones posteriores. Son patrones que no experimentan evolución ni movimiento una vez que se han formado.

Los patrones que serán mostrados a continuación, no tienen mucho por analizar, ya que son estáticos, pero más adelante se dan ejemplos en los que se muestra que pueden actuar como bloques de construcción para crear patrones más complejos.

En este proyecto se presenta la animación de una simulación (“`1_still_life.gif`”) que contiene a los siguientes patrones de esta clasificación:

1. Block

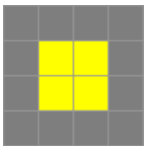

Figura 3: Estado inicial del patrón “Block” del juego de la vida de Conway.

2. Beehive

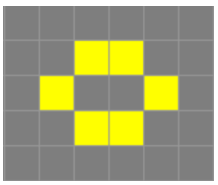

Figura 4: Estado inicial del patrón “Beehive” del juego de la vida de Conway.

3. Loaf

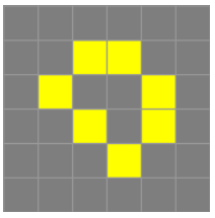

Figura 5: Estado inicial del patrón “Loaf” del juego de la vida de Conway.

4. Boat

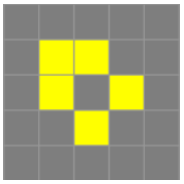

Figura 6: Estado inicial del patrón “Boat” del juego de la vida de Conway.

5. Tub

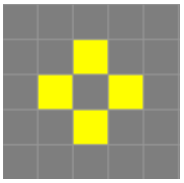

Figura 7: Estado inicial del patrón “Tub” del juego de la vida de Conway.

6. Eater 1

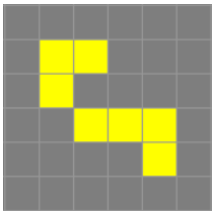

Figura 8: Estado inicial del patrón “Eater 1” del juego de la vida de Conway.

7. Bi-Block

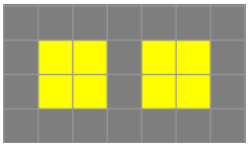

Figura 9: Estado inicial del patrón “Bi-Block” del juego de la vida de Conway.

8. Mirrored table

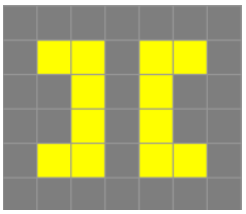

Figura 10: Estado inicial del patrón “Mirrored table” del juego de la vida de Conway.

2. Oscillator patterns

Estos patrones, son los que experimentan un ciclo repetitivo de generaciones, donde regresan a su forma original después de un cierto número de pasos llamado período. A diferencia de los "still life patterns" que permanecen estáticos, los osciladores tienen movimiento interno y cambian su configuración en cada ciclo, pero eventualmente vuelven a la misma configuración que tenían al principio.

En este proyecto se presentan diversas animaciones de patrones interesantes de analizar.

1. Short Oscillators
    a. Blinker

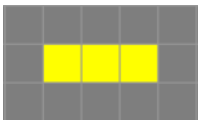

Figura 11: Estado inicial del patrón “Blinker” del juego de la vida de Conway.

    b. Toad

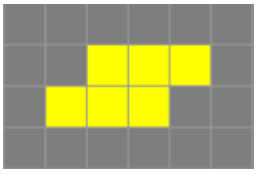

Figura 12: Estado inicial del patrón “Toad” del juego de la vida de Conway.

c. Beacon

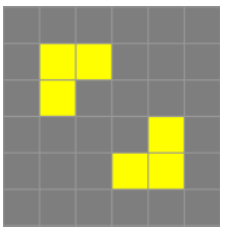

Figura 13: Estado inicial del patrón “Beacon” del juego de la vida de Conway.

Se les denominó “Short Oscillators” a aquellas estructuras de período 2, que no requieren de un análisis muy profundo, pero al igual que los “still_life”, también sirven como base de construcción para patrones más complejos.

2. Pulsar

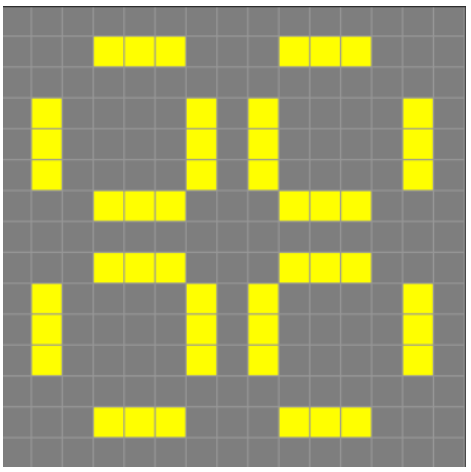

Figura 14: Estado inicial del patrón “Pulsar” del juego de la vida de Conway.

Se observó que este es un oscilador altamente simétrico de período 3, además, es uno de los más conocidos, ya que fue el primer patrón descubierto de su período, y se caracteriza por su forma de cruz con grupos de celdas vivas que parpadean de manera periódica.

3. Star

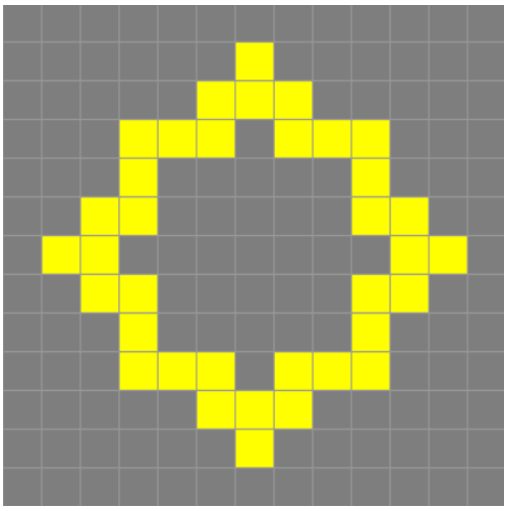

Figura 15: Estado inicial del patrón “Star” del juego de la vida de Conway.

Se analizó que este oscilador es una estructura en la que uno de sus estados es un “Polinomio”, es decir, que cada una de las células está conectada ortogonalmente. Además, existen múltiples variaciones de este patrón aunque únicamente se estudió a la más simple de ellas.

4. Pentadecathlon

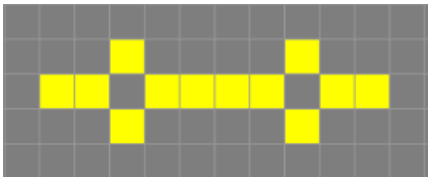

Figura 16: Estado inicial del patrón “Pentadecathlon” del juego de la vida de Conway.

El Pentadecathlon es el patrón más pequeño cuyo período es de 15 generaciones.

5. Beluchenko’s P37

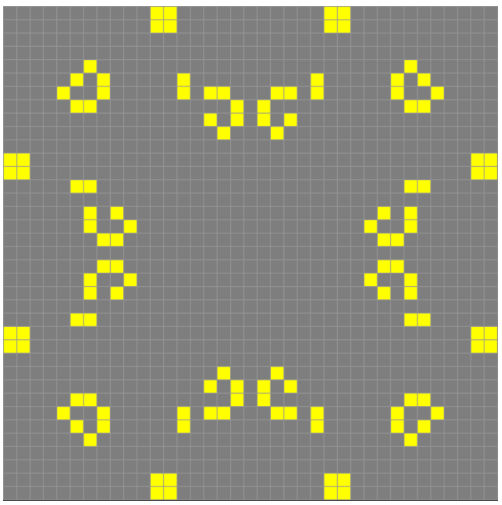

Figura 17: Estado inicial del patrón “Beluchenko 's P37” del juego de la vida de Conway.

Este patrón es el más pequeño cuyo período es de 37 generaciones, y además se observó que su estado inicial es una formación simétrica de otras configuraciones, incluyendo algunas de las mencionadas antes, como el “Block” y el “Loaf”. En el comportamiento del patrón, las figuras mencionadas recientemente cumplen la función de “contener” al desarrollo de células que se dan internamente dentro del perímetro marcado dichas estructuras.

6. 92P156

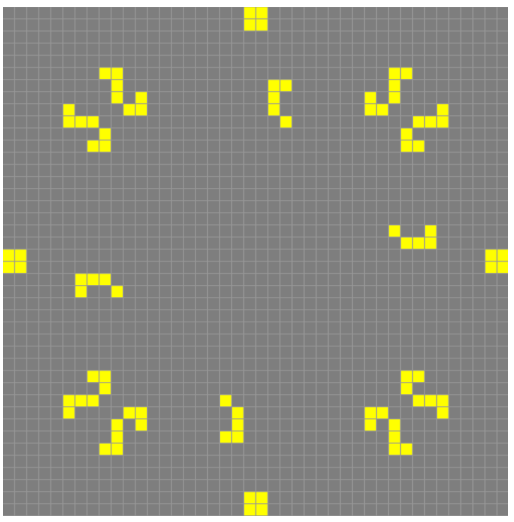

Figura 18: Estado inicial del patrón “92P156” del juego de la vida de Conway.

El “92P156” se constituye de 156 estados de generación y tiene la similitud con el anterior, en el sentido de que el estado inicial se conforma por figuras básicas; en este caso se pudo observar que un conjunto de patrones como el “Block” y el “Eater 1” cumplen la función de marcar un contorno para un desarrollo interno inicializado por otro conjunto de células.

3. Spaceship patterns

Son estructuras que se desplazan a través del tablero en una dirección constante, manteniendo su forma mientras se mueven; teniendo la característica distintiva de poder viajar a través del tablero a velocidades constantes sin desintegrarse. También tienen un período, solo que a diferencia de los “Oscillators”, el ciclo vuelve a ocurrir pero desde una posición diferente. Además de tener período, como son capaces de desplazarse, se puede medir la velocidad de desplazamiento con el número de células que el patrón mueve en cada período dividido la longitud del período.

Las animaciones que se muestran en este proyecto son las siguientes.

1. Glider

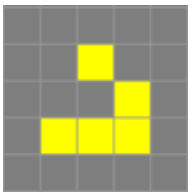

Figura 19: Estado inicial del patrón “Glider” del juego de la vida de Conway.

Se observó que este patrón cobra importancia, ya que fue una de las primeras naves espaciales descubiertas; y además es muy sencilla de formar. Más adelante se amplía sobre patrones capaz de generar estas estructuras. La estructura actual se desplaza a una velocidad de *c/4* y su período 4.

2. LWSS

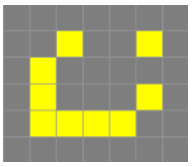

Figura 20: Estado inicial del patrón "LWSS" del juego de la vida de Conway.

Junto al patrón anterior, y el LWSS son las naves espaciales más populares. Esta configuración se desplaza a una velocidad de *c/2* y su período es 4.

3. 56P6H1V0

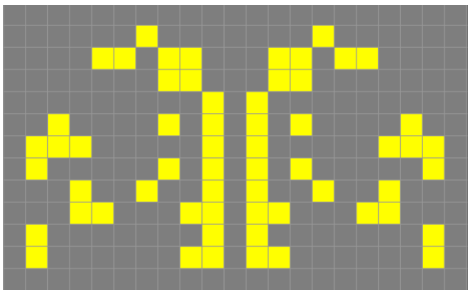

Figura 21: Estado inicial del patrón "56P6H1V0" del juego de la vida de Conway.

El período de esta nave espacial es 6, y es la más pequeña con velocidad *c/6* con dirección ortogonal.

4. 60P5H2V0

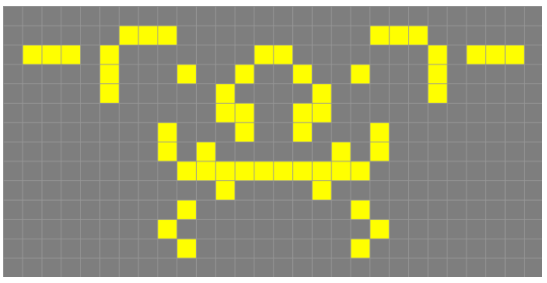

Figura 22: Estado inicial del patrón "60P5H2V0" del juego de la vida de Conway.

Se estudió que esta nave fue la primera en ser construida mediante síntesis de Gliders, es decir, colisiones de dichas estructuras para generar otras formas; pudiendo concluir a realizar masivos generadores de esta figura en particular.

4. Gun patterns

Son estructuras que generan "spaceships" de manera continua. Son patrones que emiten una secuencia periódica de naves espaciales a medida que evolucionan a través de las generaciones. Estas pistolas son fundamentales para la construcción de patrones y mecanismos más complejos en el juego, ya que proporcionan una forma de crear y transmitir información de manera constante y repetitiva.

1. Gosper’s glider gun

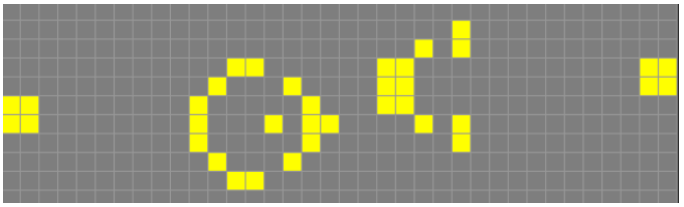

Figura 23: Estado inicial del patrón “Gosper’s glider gun” del juego de la vida de Conway.

Se observó que la Gosper's Glider Gun es una estructura relativamente grande y compleja que dispara gliders en una secuencia periódica. A medida que evoluciona a través de las generaciones, esta pistola genera un flujo constante de “Gliders” que se alejan de ella en direcciones específicas. La pistola misma es autosuficiente y puede mantener su ciclo de generación de gliders indefinidamente.

2. Period-156 glider gun

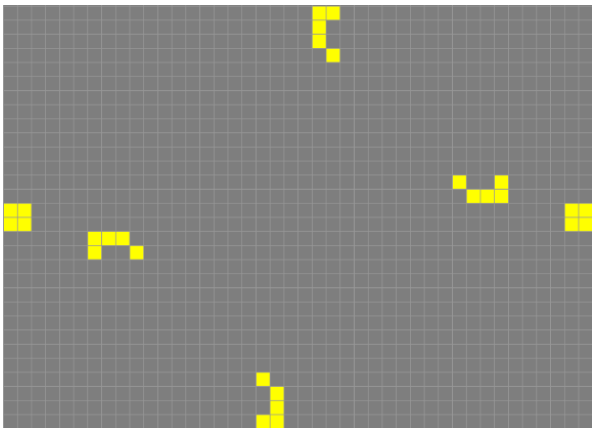

Figura 24: Estado inicial del patrón “Period-156 glider gun” del juego de la vida de Conway.

Se observó que el patrón oscilador “92P156” es una variación de esta estructura, de la cual hay diversas formas de construirla en un oscilador. En este caso, el arma dispara cuatro “Gliders” que se agrupan de a dos y van en direcciones paralelamente opuestas.