

Cálculo de *Elementary Cellular Automaton* y atractores

Sistemas Complejos

Prof: Genaro Juárez Martínez

Luis Eduardo Valle Martínez

15 de Junio del 2022

1 Introducción

El autómata celular es un modelo discreto que puede crear comportamientos complejos al usarse reglas sencillas. Su construcción se realiza utilizando una grilla, nombrada espacio, con un número finito de estados, que permite la evolución de estos utilizando una regla determinística de evolución.

La parte interesante que ha fascinado a investigadores quienes han liderado investigaciones sobre estos, es la capacidad de desarrollar un comportamiento complejo que es imposible de prever tan solo analizar la regla que se aplica.

Desarrollado el modelo por Von Neumann en los años 40 mientras trabajaba en la construcción de un sistema auto replicatorio, por consejo de su colega Stanislaw Ulam, cambió su método de aproximación utilizando una abstracción matemática, lo que llevó al desarrollo de los autómatas celulares de 2 dimensiones.

Años de investigación en la materia y después de avances en el estudio de los autómatas bidimensionales, se empezó a estudiar a los autómatas celulares en su forma más simple reduciéndolos a 1 dimensión, cambiando las vecindades posibles de 4 u 8 vecinos a 2, llamándoles comúnmente como *Elementary Cellular Automaton*(ECA).

Al igual que sucede con los autómatas bidimensionales, las células tienen 2 estados con la posibilidad de interacción con el par de células vecinas $i - 1, i + 1$. De estas vecindades nacen las reglas que permiten su evolución, y se definen como una función matemática que asigna a un número de 3 bits un número de 1 bit el cual corresponde al nuevo estado de la célula central.

Dado que existen 8 posibles configuraciones de los vecindarios compuestos de 3 células, existen un total de $2^8 = 256$ ECAs.

Dado que los ECA definen sus espacio unidimensional como una cinta enlazada en sus extremos, el comportamiento del sistema a través de las generaciones de evolución se consigue utilizando diagramas de espacio tiempo, en el cual la configuración de los estados en la rejilla d dimensional, es graficado como una función del tiempo.

Los ECAs son particularmente interesantes de analizar y estudiar por 2 razones, la primera de ellas es el número limitado de reglas y sus interacciones las vuelven sencillas de estudiar en comparación por ejemplo de los autómatas bidimensionales. El segundo tema de interés, es la naturaleza visual del tiempo permite realizar investigaciones profundas en los patrones cambiantes que se encuentran en sus evoluciones.

Con estas características los ECAs se han convertido en una herramienta para la exploración de la aparición(*emergence*), caos y complejidad en un sistema no lineal.

En este trabajo se plantea la creación de un simulador de ECAs que permita observar gráficamente mediante una animación de una grilla con células, las evoluciones de las vecindades al definirse una regla de evolución. Así también se proveen demás funcionalidades en una interfaz gráfica que permite cargar configuraciones en un espacio vacío, guardar el espacio de evolución en un archivo, realizar un análisis estadístico de las generaciones con la graficación de la densidad y entropía de Shannon, y finalmente la capacidad de calcular los campos de atracción de una regla específica dado un rango de potencias.

2 Programa

2.1 Dependencias

Lista de dependencias de bibliotecas y programas requeridos para la ejecución del programa. Se utilizó para el desarrollo del simulador el lenguaje de programación Python en su forma vanilla, sin uso de algún *framework*, por lo que mínimamente se requiere la instalación de Python en su versión 3. El desarrollo se realizó en la versión 3.9.7.

Las bibliotecas listadas a continuación pueden instalarse mediante programas como pip, o directamente en un ambiente virtual como *pipenv* o *anaconda*:

- PyGame 2.1.2
- Numpy 1.21.5
- Matplotlib 3.5.1
- Networkx 2.8.3
- igraph 0.9.10
- cairocffi 1.0.0

PyGame es una biblioteca utilizada para el desarrollo de sencillos juegos 2D en python, incluyendo algunas herramientas gráficas para la impresión en pantalla de diferentes formas geométricas, imágenes y *Sprites*. Incluye métodos usados para la detección de colisiones entre objetos, impresión de áreas en pantalla y otros de configuración como el número de veces de refresco de pantalla en cuadros por segundo FPS(*Frames Per Second*). Incluye así también métodos para la fácil implementación de sonido y archivos de audio, sin embargo este tipo de funciones no se utilizan en el desarrollo de este simulador.

Numpy es un biblioteca para el manejo de vectores y matrices, incluyendo un amplio conjunto de recursos en funciones matemáticas en el dominio del álgebra lineal, transformada de Fourier y matrices. Principalmente el uso de esta biblioteca en el programa se encuentra en el uso de *arrays* como estructuras para el manejo de espacios y otros recursos auxiliares que permiten de forma sencilla y eficiente las operaciones durante la evolución del espacio de los autómatas.

Networkx es una biblioteca de Python enfocada en el estudio de grafos y redes. Permite la creación, manipulación y estudio de las estructura, dinámicas, y funciones de redes complejas.

igraph es una colección de bibliotecas para la creación y manipulación de grafos y análisis de redes. Escrita en C inicialmente, se han creado paquetes que funcionan en Python y R.

cairocffi es un reemplazo directo basado en CFFI para Pycairo, un conjunto de enlaces de Python y una API orientada a objetos para cairo. Cairo es una biblioteca de gráficos vectoriales 2D con soporte para múltiples backends, incluidos búferes de imágenes, PNG, PostScript, PDF y salida de archivos SVG.

La totalidad del proceso de desarrollo y pruebas del funcionamiento de la simulación se realizó utilizando un SO llamado Pop!_OS en su versión 21.10, sistema basado en UNIX Linux derivado de la popular distribución Debian.

La ejecución del programa en otro sistema operativo diferente a una distribución Linux no debería ser impedida si se cuenta con la instalación de todas las dependencias, ya sea en un ambiente virtual o el espacio de trabajo de usuario directamente.

2.2 Requerimientos

El programa construido consiste en un simulador gráfico de un espacio de evolución para una cierta configuración de Autómatas Celulares Elementales en una única dimensión, basado en el trabajo desarrollado por Stephen Wolfram. La evolución de los autómatas se basa en los valores de una vecindad que considera 3 células en la misma fila de evolución, dependiendo de la configuración de estas 3 células la siguiente generación en la célula central está dada por el valor del bit en la posición representada por la configuración de la vecindad sobre la cadena binaria de la regla especificada por el usuario. Al utilizarse una vecindad de 3 reglas el número de configuraciones es igual a 8, así directamente se puede hacer una relación de cada regla con 1 bit en un byte, al asignarsele un valor a la regla entonces el estado del bit para cada configuración(0 o 1) será el valor de la célula central en la siguiente evolución(Muerta o Viva).

Para facilitar el uso del simulador y precisamente poder observar la evolución de una manera intuitiva en forma de animación, el programa requirió de una implementación gráfica, aprovechándose esta cualidad para incluir botones o diferentes componentes que permitan al usuario configurar los parámetros y condiciones en el que tendrá lugar el proceso de evolución para la configuración del espacio.

A continuación se numeran los requerimientos funcionales con las que el programa debe cumplir:

- Ingresar las potenciales 256 reglas de evolución por el usuario.
- Evaluar espacios de hasta 1000x1000 células. Alternativamente si no es posible mostrar en una sola pantalla la totalidad del espacio implementar un scroll de pantalla.
- Proporcionar el cambio de colores en las células según su estado.
- Poder inicializar el espacio de evoluciones con diferentes densidades. Alternativamente el uso de archivos para una configuración inicial o modificando manualmente el espacio de evolución.
- Agregar la funcionalidad de guardado y levantado de archivos con configuraciones previamente calculadas.
- Habilitar la graficación de la densidad por generación, graficación del logaritmo de la densidad por generación y finalmente la entropía del espacio de evolución por generación.
- Calcular los atractores resultantes en una configuración utilizando un conjunto de universos binarios potencia especificados como un rango por parte del usuario, para una regla de evolución.

2.3 Interfaz

La biblioteca gráfica principal para la construcción de la GUI del simulador fue *PyGame*.

Las herramientas que provee tal biblioteca son insuficientes y en muchos de los casos para típicos elementos gráficos de una GUI(botones, *sliders*, campos de texto, etc), son inexistentes. Por esta razón se considera un biblioteca de bajo nivel, y que dentro de sus capacidades se enfoca en facilitar tan solo algunas implementaciones más relacionadas con elementos gráficos y funcionales de sencillos juegos en 2D, que como una biblioteca para el desarrollo de interfaces.

Debido a la naturaleza de la biblioteca se presentó como un reto la construcción de la interfaz, explicándose posteriormente el enfoque abordado para solucionar estas limitaciones, codificando los componentes gráficos desde 0 y en un ambiente de manejo de posicionamiento por coordenadas de pixeles con algunas figuras geométricas básicas como círculos o rectángulos.

Habiéndose creado e implementado los elementos básicos gráficos para una GUI, junto con otras implementaciones de organización y posicionamiento gráfico(*layouts*), la interfaz final es como la siguiente(Figura 2.3)

2.3.1 Componenentes Gráficos

Todos los componentes gráficos que se describen pertenecen al módulo de la aplicación nombrado *GraphicalComponents* y que puede verse su implementación Python en la sección de **Código Fuente**.

Botón Los botones son componentes gráficos básicos en cualquier GUI, y no es excepción en el programa de simulación pues se utilizan un total de 12 botones y que permiten desempeñar diferentes acciones.

La primer triada de botones son los utilizados para Pausar, Correr y Cargar una nueva configuración de la simulación. Ubicados en la parte superior derecha en la sección lateral, son de los botones más importantes que ejecutan las funciones básicas de la evolución.

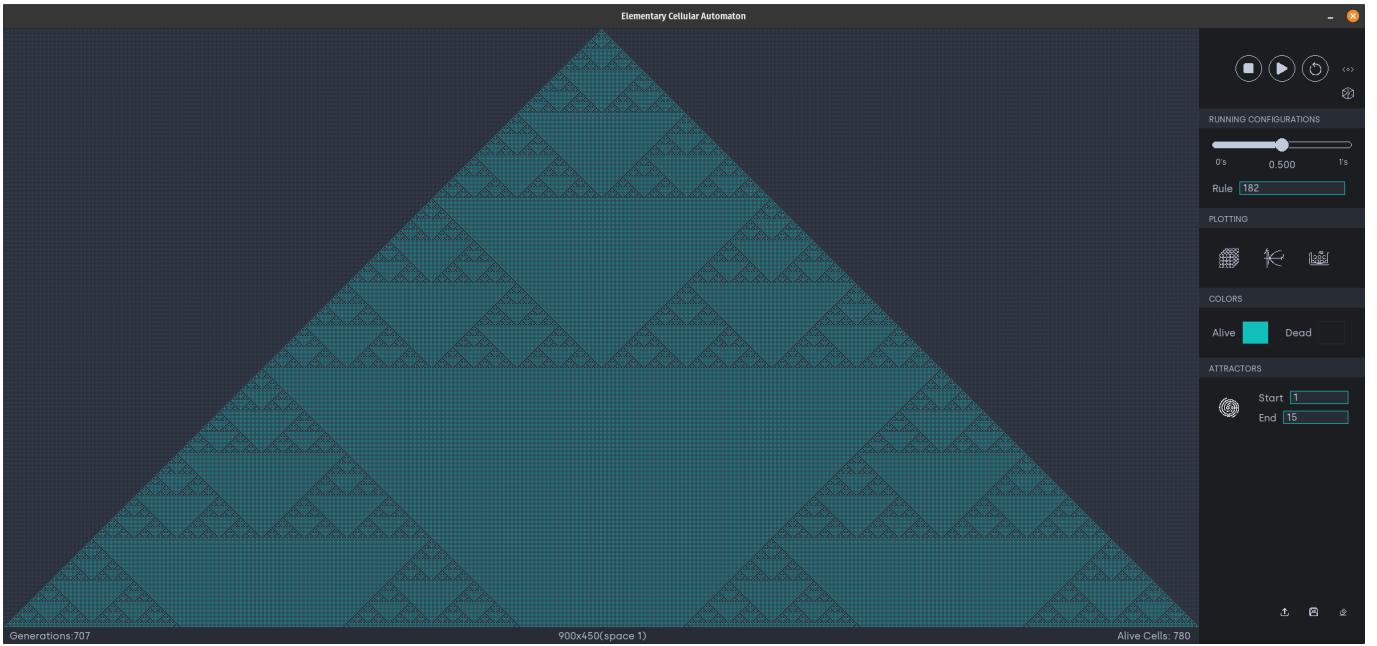


Figure 1: Captura de pantalla de la interfaz gráfica del simulador al ejecutarse



Figure 2: Botones que dictan el flujo de las evoluciones del espacio

Ubicados también en la sección lateral en su parte superior, se encuentra el par de botones que permiten elegir una configuración inicial. El primero de ellos coloca una única célula viva justo en el medio del espacio de evolución en la fila inicial. El segundo de ellos, representado por un dado, inicializa la primer generación con un número dependiente de la densidad especificada con el slider de células vivas y muertas con una configuración aleatoria.

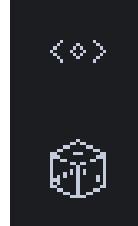


Figure 3: Permiten rápidamente colocar una configuración inicial para la evolución del simulador

Siguiendo en importancia ubicados en la parte inferior de la sección lateral se ubican los 3 botones de acciones relacionadas al espacio gráfico de evolución. El primero de ellos representado por un ícono de una flecha es el botón *Charge* y permite cargar una configuración en un archivo CSV. Le sigue el botón con ícono de disquet *Load*, que permite guardar la actual configuración del espacio de evolución. Finalmente se tiene un ícono de goma de borrar *Clear*, que limpia el espacio de evolución colocando el espacio con todas las células muertas.

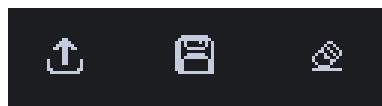


Figure 4: Botones inferiores para la configuración del estado de evolución

Finalmente ubicadas en la sección de *Plotting*, se tienen 3 botones que se encargan de la graficación. El primero de ellos con un ícono de cubo corresponde a la gráfica de Densidad para las generaciones corridas. Le sigue un botón con ícono de una gráfica logarítmica, y que precisamente grafica el logaritmo de las densidades. Finalmente se tiene el ícono con unas partículas de un gas para la graficación de la Entropía.

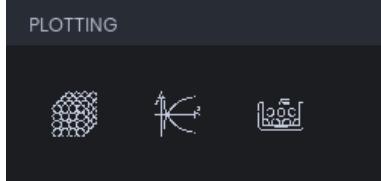


Figure 5: Botones con íconos para la graficación de la complejidad de los estados evolucionados

Texto Se utilizan algunos textos para indicar el estado actual del espacio de evolución, siendo los 3 más importantes aquellos impresos en la barra inferior, y que respectivamente indican: El número de generaciones que han transcurrido, El tamaño del espacio y entre paréntesis el número de pantalla actual(pantallas mostradas por el scroll), y finalmente el número de células en el espacio que se encuentran vivas.



Figure 6: Textos ubicados en la barra inferior. Con el propósito de mostrarlos los 3 la imagen se ha recortado para ubicarlos juntos, sin embargo en la interfaz se muestran en las esquinas y el centro de la ventana en la parte inferior

Slider Un slider es un componente gráfico algo más complejo que va a estar formado por una combinación de los 2 elementos anteriores así como un par de rectángulos y círculos que le dan forma al cuerpo de la ranura por la que el botón se desliza.

La función de este slider es la elección del complemento de la densidad, y que es aplicado cuando se da al botón Reset que carga una nueva configuración de espacio de evolución con un número aleatorio de células vivas que se rigen por el complemento de la probabilidad que se escoge con el slider.



Figure 7: Slider para elegir la probabilidad de las células muertas cuando se realiza un Reset

Input El input es un elemento típicamente utilizado en formularios para el ingreso de texto corto. El input cuenta con una *Label* etiqueta que sirve al usuario para indicar el tipo de respuesta que se debe colocar ahí, así como del cuerpo del input y el texto por supuesto.

La especificación de la regla de evolución por parte del usuario se implementa como un input en el que se ingresa el número de la regla. Típicamente el valor del input por defecto será 110(regla de especial interés en los ECAs)

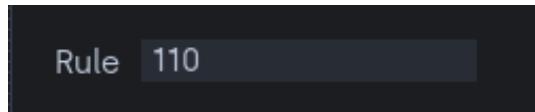


Figure 8: Input utilizado para el ingreso de la regla de evolución

Un uso poco convencional pero conveniente, es la implementación de un par de inputs en forma de una clase de botones que permiten el cambio de los colores para las células vivas o muertas.

Se optó por utilizarse estos elementos por el texto de etiqueta que por defecto se coloca en el input, colocándose un texto vacío e ignorando eventos de tecla que los pueda modificar, siendo únicamente el click el que desencadene la función para el cambio de color.

Los colores disponibles pueden consultarse en el módulo *Constants* en la sección **Código Fuente**.

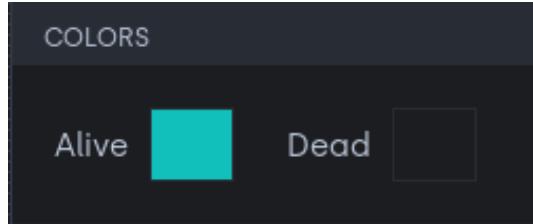


Figure 9: Par de inputs ubicados en la sección *Colors* y que permiten modificar el color actual de las células vivas y muertas

Finalmente, pero no menos importante, el uso conjunto de inputs y botones en la interfaz se encuentra en la sección *Attractors*. A través del par de inputs se indica el rango de potencias del universo binario con las que se ejecutará el proceso de cálculo de los atractores.

Por último el botón ubicado en la parte izquierda se presiona para empezar el proceso de cálculo, mostrándose inmediatamente los árboles de forma gráfica en el espacio de evolución y graficados como imágenes en la carpeta *graphs*.

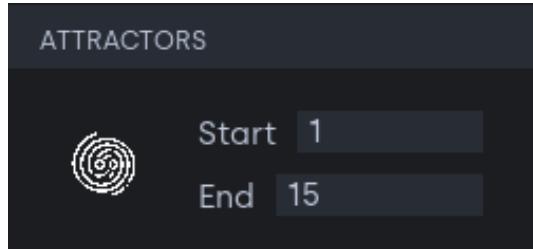


Figure 10: Botón e inputs utilizados para la ejecución del proceso de cálculo de los atractores de una regla

Células Gráficas Este componente gráfico es el único que no se encuentra en el módulo *GraphicalComponents* pues se encuentra declarado en el módulo *Graphics*, esto se debe a que a diferencia de los elementos anteriores las células no son componentes comunes en una biblioteca GUI de propósito general, siendo particulares de esta aplicación de simulación.

Las células gráficas son en esencia un *sprite* en forma de rectángulo que cambiará su color en los diferentes casos en función de si esa célula se encuentra viva o muerta:

- Se presiona o pasa encima el cursor mientras se mantiene presionado el click para cambiar el estado de la célula y por lo tanto su color.
- Durante el proceso de evolución la célula cambia su estado
- De forma general todas las células cambian a muertas cuando se da un *Clear*.
- Dependiendo de la configuración inicial aleatoria algunas células pueden cambiar su color pues se han definido como células vivas.

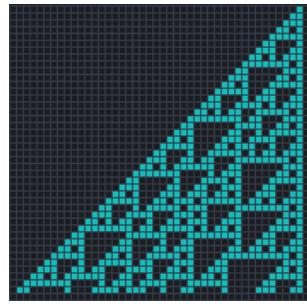


Figure 11: Fragmento de células gráficas en el espacio de evolución

Section Componente con representación gráfica en forma de una franja que contiene un encabezado con el título de la sección.

Este recurso se utiliza para realizar la separación visual de los botones de graficación y los colores de las células, pero también se utiliza como referencia para colocar los elementos gráficos clasificados dentro de esa sección.

2.3.2 Layouts

Como se explicó previamente, el posicionamiento de los elementos en la ventana se realiza a nivel coordenadas de pixel, por lo que para algunos casos como la colocación de las células gráficas o las secciones y barras laterales e inferior, se utilizan componentes de *layout* u organización, que pueden o no tener alguna representación gráfica.

Estos objetos pueden encontrarse en el módulo *Layouts*

SideBar Componente *singleton*, instanciado 1 sola vez en la ventana, que tiene representación gráfica como un rectángulo en la parte lateral derecha de la ventana.

Semánticamente separa a la sección del espacio de evolución para albergar en su interior los diferentes componentes que controlan la evolución del simulador.

BottomBar Componente *singleton*, instanciado 1 sola vez en la ventana, que tiene representación gráfica como un rectángulo en la parte inferior de la ventana.

Semánticamente separa a la sección del espacio de evolución y la barra lateral, para albergar en su interior los textos que indican el estado de la evolución así como el tamaño y zoom.

Grid Componente sin representación gráfica que fue desarrollado para mimetizar el comportamiento de este tipo de organización gráfica en forma tabular.

Este objeto se encarga de calcular el tamaño de sus elementos según las medidas proporcionadas para ocupar todo el grid, y los paddings entre cada elemento.

Se le pasa una lista de los Rectángulos de los elementos gráficos para que según la disposición calculada automáticamente se coloquen los elementos en su respectiva posición, conformando finalmente el grid completo con las dimensiones y número de filas y columnas indicadas.

2.4 Espacio de evolución

Dos espacios de evolución se ocupan para la construcción del programa simulador. El primero de ellos representante del espacio gráfico, es visible en un arreglo tipo *grid* y se conforma de los componentes gráficos nombrados como **Células Graficas** en el contexto del programa.

En este espacio de evolución el usuario puede desarrollar directamente tan solo la configuración manual del estado de cada célula, acotándose esta función únicamente a la fila que corresponde a la generación actual, filas previas y posteriores son imposibles de modificar.

Manteniendo una estrecha relación de correspondencia se tiene un segundo espacio de evolución, este espacio como recurso lógico de las células, se implementa para esta solución como un espacio de tamaño ($NUMBER_CELLS_1D + 2) \times 2$. La primer fila del espacio se corresponde a la configuración actual de la fila que muestra la última generación calculada, mientras que la segunda se utiliza para guardar la configuración mientras se realiza el cálculo, de la siguiente generación tras la evolución. Una vez este cálculo se realizó el valor de la segunda fila pasa a la primera y se limpia esta última hilera.

Es importante notar que la dimensión del espacio lógico tiene añadido un par de columnas, las cuales se implementan como un padding que permite replicar el mecanismo de una cinta enlazada, y que es sumamente útil cuando se particionan las vecindades para el cálculo de la nueva generación.

Complementario a este par de espacios, se cuenta con uno más especificado como un arreglo numpy utilizado para guardar los valores de las generaciones anteriores.

2.4.1 Configuración del espacio de evolución

En el archivo Python nombrado como *main.py* se tienen un conjunto de constantes de configuración que pueden ser modificadas para cambiar los aspectos gráficos y lógicos de la simulación, no siendo la excepción el espacio de evolución donde los valores de las siguientes constantes influyen en su construcción:

- **GRID_WIDTH:** Especifica el tamaño en pixeles del ancho del grid en el que se acomodan las células gráficas. Es importante que este valor sea al menos igual al **NUMBER_CELLS_1D** para tener células de 1px, de otra forma se da un error pues no se pueden crear células gráficas con tamaño menor al pixel.
- **GRID_PADDING:** En el aspecto gráfico, esta constante se modifica para añadir espaciado entre célula y célula en el grid
- **NUMBER_CELLS_1D:** Indica el número de células que se tendrá por generación. Gráficamente se traduce en el número de columnas por cada fila.
- **NUMBER_EVOLUTIONS_GRID:** Indica el número de generaciones que se mostrarán gráficamente en una sola pantalla, que es lo mismo al número de filas para el espacio de evolución.

2.4.2 Configuración única célula central

Particular a los ECAs, la evaluación de un espacio de evolución se realiza con una única célula viva en el centro de las columnas disponibles. Esta configuración de célula solitaria permite identificar correctamente el proceso de evolución así como los patrones que genera cada una de las reglas.

Por esta razón se consideró e integró a la simulación esta opción como la primera de las 3 existentes.

2.4.3 Configuración aleatoria inicial

La configuración aleatoria inicial consiste en definir una configuración nueva en el espacio de evolución de la primer fila, basándose en la probabilidad de densidad para que el nuevo estado de la célula sea viva. Esta probabilidad de densidad es configurable directamente utilizando el *slider* gráfico(Figura 7), iniciando en primera instancia inmediatamente después de correr la simulación en 50% para ambos estados.

Si se desea cargar una nueva configuración aleatoria inicial, aún después de haber iniciado un proceso de evolución, se puede dar click al botón de reset(Figura 2).

2.4.4 Configuraciones personalizadas

También es posible configurar de manera particular la configuración de inicio, así como la configuración de cada una de las células correspondientes a las células en la fila de la generación actual.

La mecánica se ha explicado anteriormente, pero consiste en con el mouse dar click o arrastrar, sobre las células para cambiar su estado. Si estas se encuentran vivas al colapsar con el puntero cambiarán a muertas, sucediendo también para cuando se encuentran muertas y renacen.

Aún cuando se integra esta mecánica al simulador, no resulta como una herramienta de gran utilidad como si sucedía en el simulador bidimensional, donde era primordial para la creación de configuraciones con estructuras definidas, y de esta forma evaluar su comportamiento. En este caso al tratarse de 1 sola dimensión, el comportamiento se evalúa con generaciones que van respecto al tiempo, por lo que los colapsos o demás mecánicas plenamente apreciables en el tipo Juego de la Vida, suelen requerir aproximaciones distintas a las planteadas en el simulador anterior.

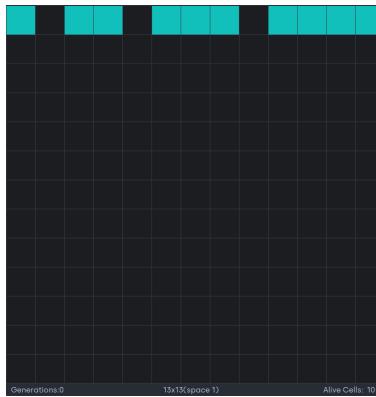


Figure 12: Ejemplo de una configuración personalizada en un espacio de 13x13 con una cuenta ascendente en la primer fila para la configuración inicial

2.4.5 Limpiar, Guardar o Cargar espacio de evolución

Controlado por la triada de botones en la parte inferior derecha(Figura 4), permite a la simulación limpiar completamente el espacio de evolución cambiando el estado de todas las células a 'Muertas' en ambos espacios(gráfico y lógico) y colocando en 0's el valor de las generaciones y células vivas.

El par de botones restantes son el complemento el uno del otro de una acción. La acción de guardar el espacio de evolución va a generar un nuevo archivo con extensión CSV guardándose en la carpeta *saves*. Este archivo nuevo representa en 0's y 1's el estado del espacio de evolución actual, razón por la cual si se cuenta con un archivo con estas características se puede cargar de vuelta en una simulación.

El proceso de cargado de un espacio guardado requiere que dentro de la carpeta *saves* se tenga el archivo con la configuración deseada renombrado a *upload.csv*, siendo imposible por el momento la elección específica de otro archivo e incluso generando un error si se acciona el botón de *Upload* sin existir este archivo en la carpeta. Este proceso permite cargar configuraciones de espacios de evolución iguales o más pequeños que el espacio actual, centrándose la configuración cargada con respecto columnas pero colocándose siempre desde la fila inicial cuando se trata con configuraciones más pequeñas que el actual espacio de evolución(Figura 13).

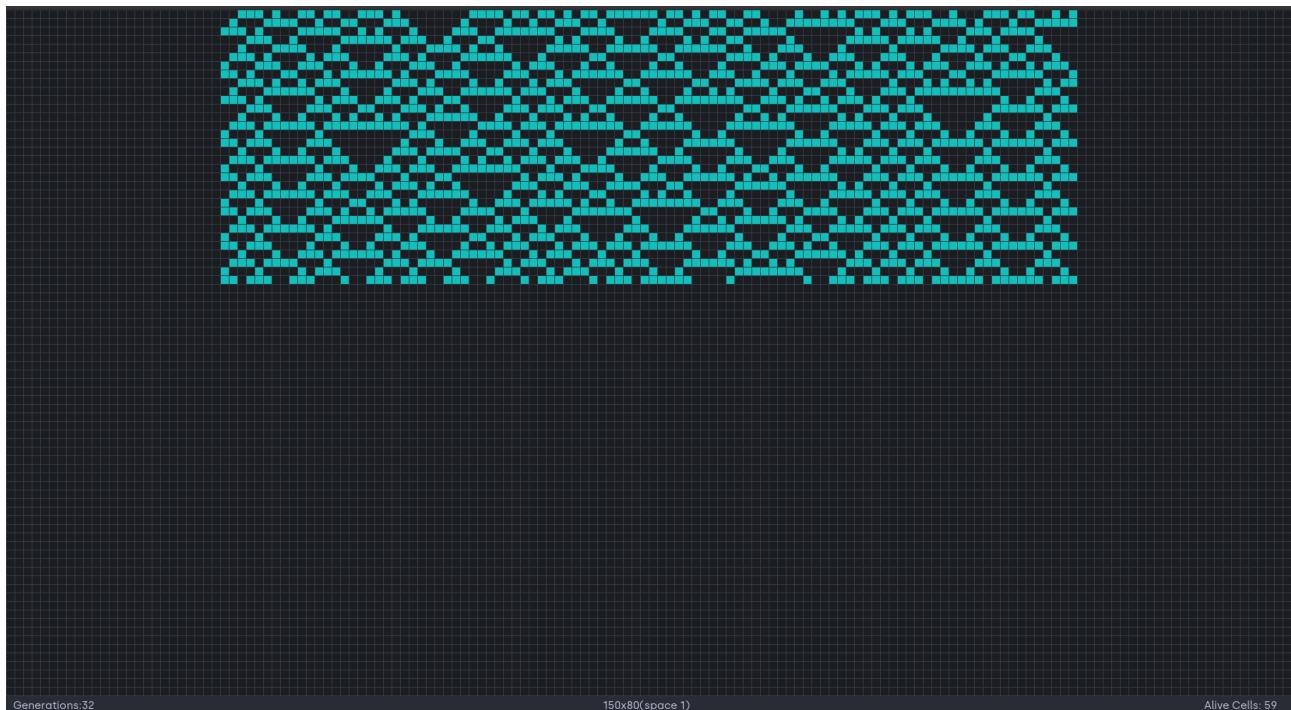


Figure 13: Ejemplo del centrado de una configuración de tamaño 100x32 cargada en un espacio de evolución 150x80

2.4.6 Cambio de colores

La simulación permite cambiar los colores de las células gráficas de ambos estados, contando para esto de un par de componentes gráficos(Figura 9) a los cuales al dar un click sobre estos cambiará su color, tanto el recuadro del input como el grid de las células gráficas que actualmente tengan el estado modificado, por el siguiente en la lista llamada *COLORS_LIST* en el módulo *Constant*. Esta lista no es más que un conjunto de 13 tuplas con 3 elementos cada una, representando cada tupla un color en formato RGB.

Por el momento el cambio de colores solo se logra de esta forma por lo que no es posible asignar un color específico no identificado en el módulo, sin embargo esta cantidad de colores generan una buena cantidad de combinaciones suficientes para un programa simulador de este tipo.

Ejemplos del cambio de colores se notan en las figuras: 14,15,16 y 17

2.4.7 Proceso de evolución

La función para el cálculo de la siguiente generación se encuentra en el módulo *ECA* de la sección *Código Fuente*.

El proceso del cálculo de una nueva generación consiste en la identificación de la configuración de una vecindad compuesta por 3 células. Tomando a una célula viva como 1 y a una célula muerta como 0, se puede tomar como una cadena binaria de 3 bits y que su valor decimal será el número del bit en la cadena binaria de la regla, del que tomará valor la célula ubicada en la columna central(columna de la 2da célula) de estas 3 en la siguiente generación.

En comparación con su contraparte bidimensional, los recursos requeridos para el cálculo de una siguiente generación suele ser mucho más económico, permitiendo trabajar de forma fluida y con espacios de evolución grandes sin la necesidad de implementar un mecanismo de computación paralela como sucede con la programación CUDA.

Esta función es el primero de los algoritmo más relevante en toda la simulación, y que requiere de un par de recursos para ser calculada:

- Espacio de anillo: Este recurso describe el comportamiento del espacio de evolución durante el cálculo de una nueva generación, en el que al igual que un anillo, enlaza los extremos de la fila del espacio. Su integración permite asegurar la existencia de vecindades completas en cada una de las células, siendo específicamente beneficiadas las células que existen en los extremos de las filas
Existen algunas alternativas para lograr este comportamiento, más la solución utilizada en este trabajo fue la adición de un *padding* en las filas como un par de elementos extras en los extremos del *array* de evolución, y sobre los cuales se replican los valores de los extremos contrarios.
- Regla de evolución: Se define como un número entero entre el rango de [0, 255]. La elección de este rango se debe al número de combinaciones distintas posibles para una cadena binaria de 8 bits, longitud originada del número de todas las posibles configuraciones de vecindad de 3.

Este proceso se lleva a cabo en una función que itera sobre las columnas del *array* de evolución, no tomando en cuenta la longitud total con los paddings, pero si considerándolos por las razones expuestas anteriormente.

Se conforman vecindades creando ventanas de células con tamaño 3 y se procede aplicar el algoritmo mencionado anteriormente. Convirtiendo la vecindad a una cadena binaria de tamaño 3, se transforma a su correspondiente valor decimal, y de esta forma se obtiene el índice(número de bit) de la cadena binaria que describe a la regla que previamente ingresó el usuario como un número decimal.

Tomando como ejemplo la regla 30, las configuraciones de vecindad que en su siguiente generación tendrán un predecesor son: '100', '011', '010', '001'. Esto es sencillo de visualizar cuando identificamos que los bits número 4,3,2 y 1, se encuentran activos en la representación binaria del número 30.

Los nuevos valores de célula correspondientes a la nueva generación son almacenados temporalmente en la segunda fila del *array* de evolución, esperando a remplazar en la primer fila a los valores anteriores tan pronto termine de recorrerse todas las vecindades en la banda. Los valores que anteriormente conformaban la primer fila del *array* de evolución, ahora serán almacenados en el arreglo del historial del espacio de evolución. Así mismo este cambio de nueva generación realizará una impresión cambiando los estados de las células gráficas en la última fila sin haber evolucionado antes, respetando la integridad entre valores de los espacios gráficos y lógicos.



Figure 14: Evolución con regla 62 desde una configuración con célula central para un espacio de tamaño 300×150 y con colores amarillo(vivas) y negros(muertas)

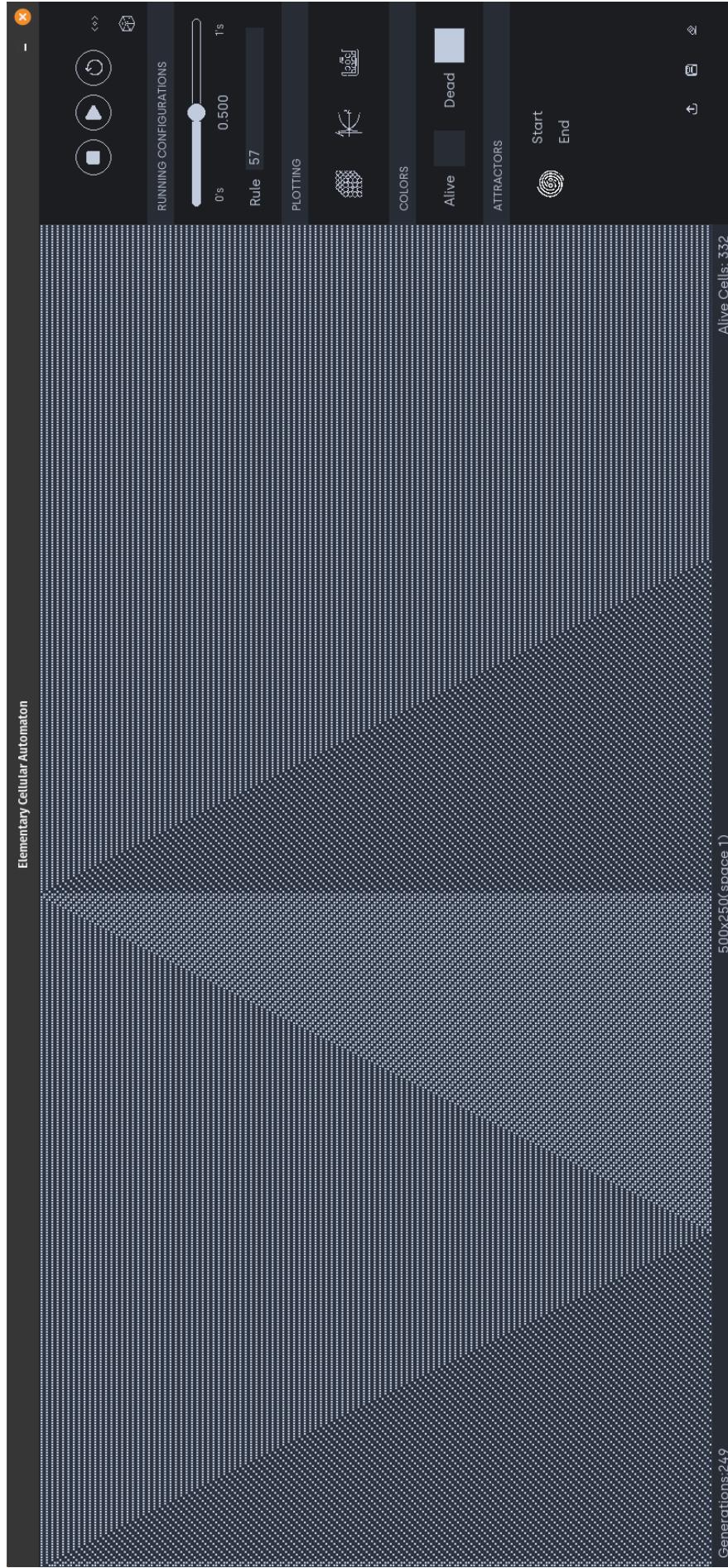


Figure 15: Evolución con regla 57 desde una configuración con célula central para un espacio de tamaño 500×250 y con colores negro(vivas) y blanco(muertas)

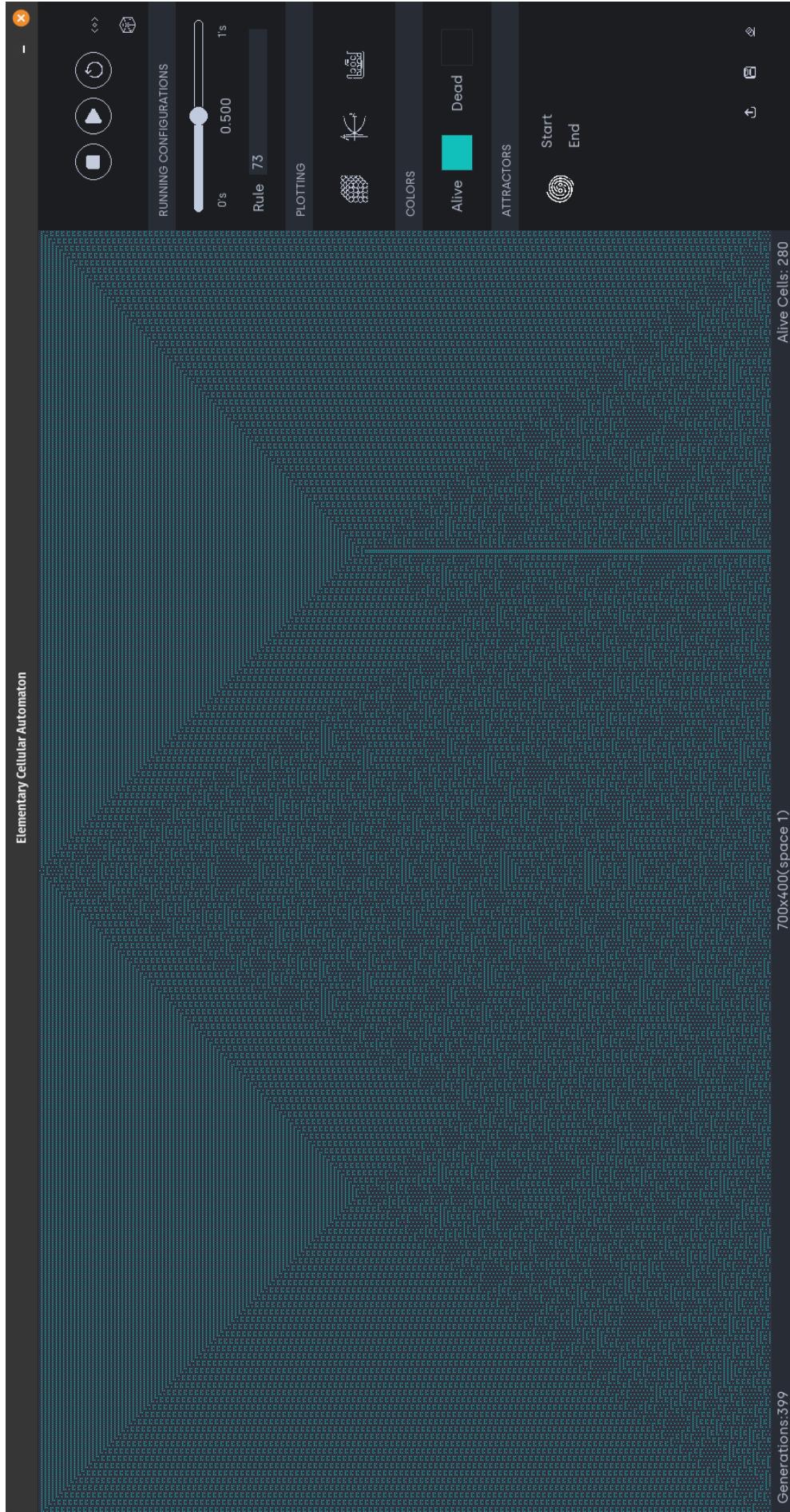
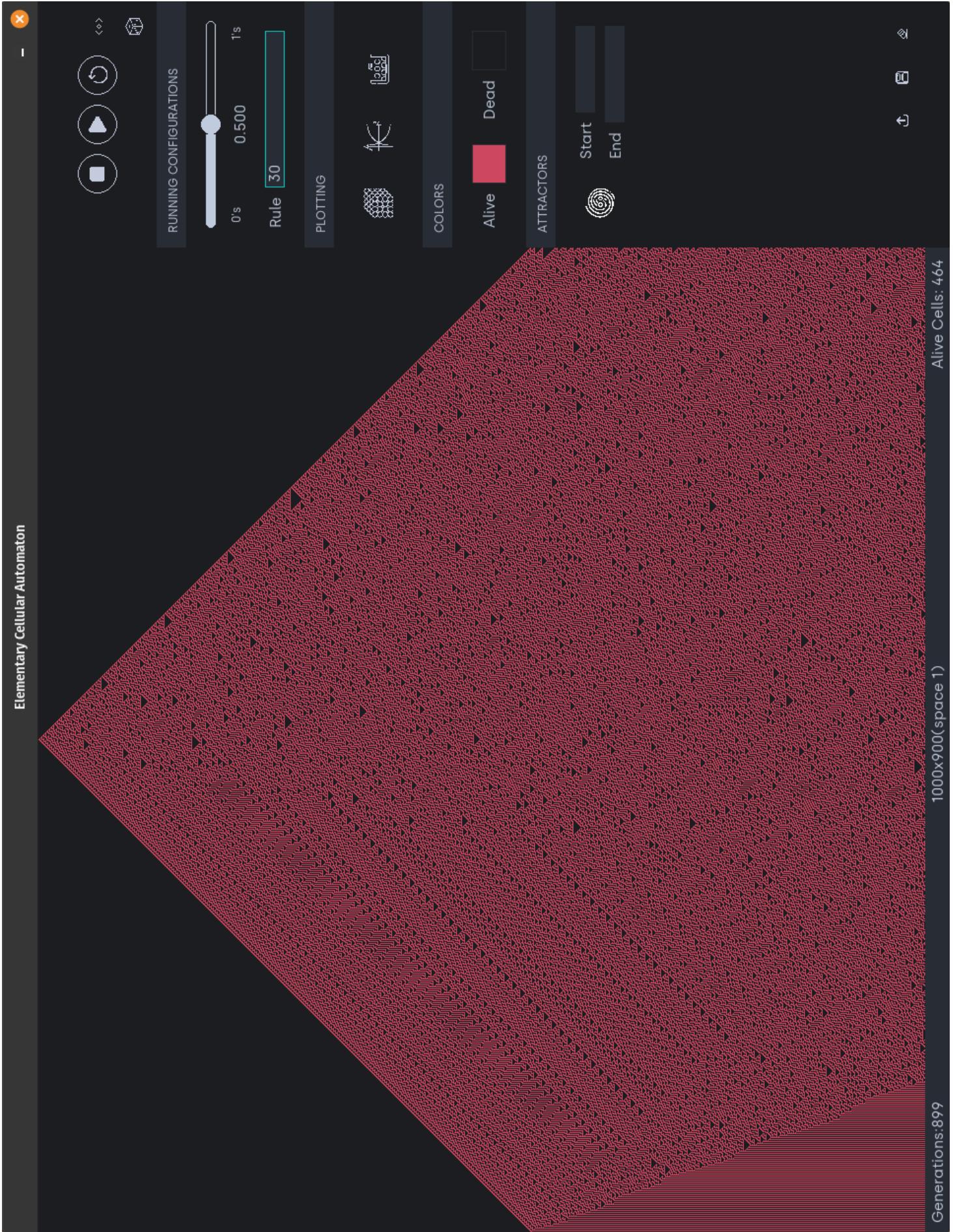


Figure 16: Evolución con regla 73 desde una configuración con célula central para un espacio de tamaño 700×400 y con colores azul(vivas) y negro(muertas)



2.4.8 Scrolling

Considerando la experiencia adquirida después del desarrollo del primer simulador en un ambiente bidimensional, se prestó especial atención en mejorar el desempeño del simulador para correr espacios de evolución mucho mayores.

Con las consideraciones tomadas para este simulador, y también gracias al disminuido gasto de recursos computacionales por tratarse de 1 dimensión, este programa es capaz de correr espacios mayores al mínimo requerido de 1000x1000, y más sin embargo en la gran mayoría de monitores esto es poco viable pues aún con células de 1px por 1px, los demás elementos gráficos como la barra inferior y la barra de la ventana definida por el sistema acortan el área visible de la ventana.

Una enfoque que se tomo para solucionar el problema de espacios de evolución pequeños fue la implementación de un *scroll* vertical. Esta mecánica permite al usuario calcular potencialmente infinitas generaciones de evolución, realizando el cambio automático entre pantallas cuando una configuración alcanza la última fila de generación visible para mostrar una siguiente pantalla limpia donde seguirá evolucionando sin contratiempo(Figura 18).

Para navegar entre pantallas basta con utilizar la rueda del mouse girando hacia arriba para una pantalla anterior, o hacia abajo para una pantalla posterior. En el caso de computadoras portátiles, gestos hacia cada dirección vertical con 2 dedos sirve como sucedáneo a la carencia de un tercer botón.

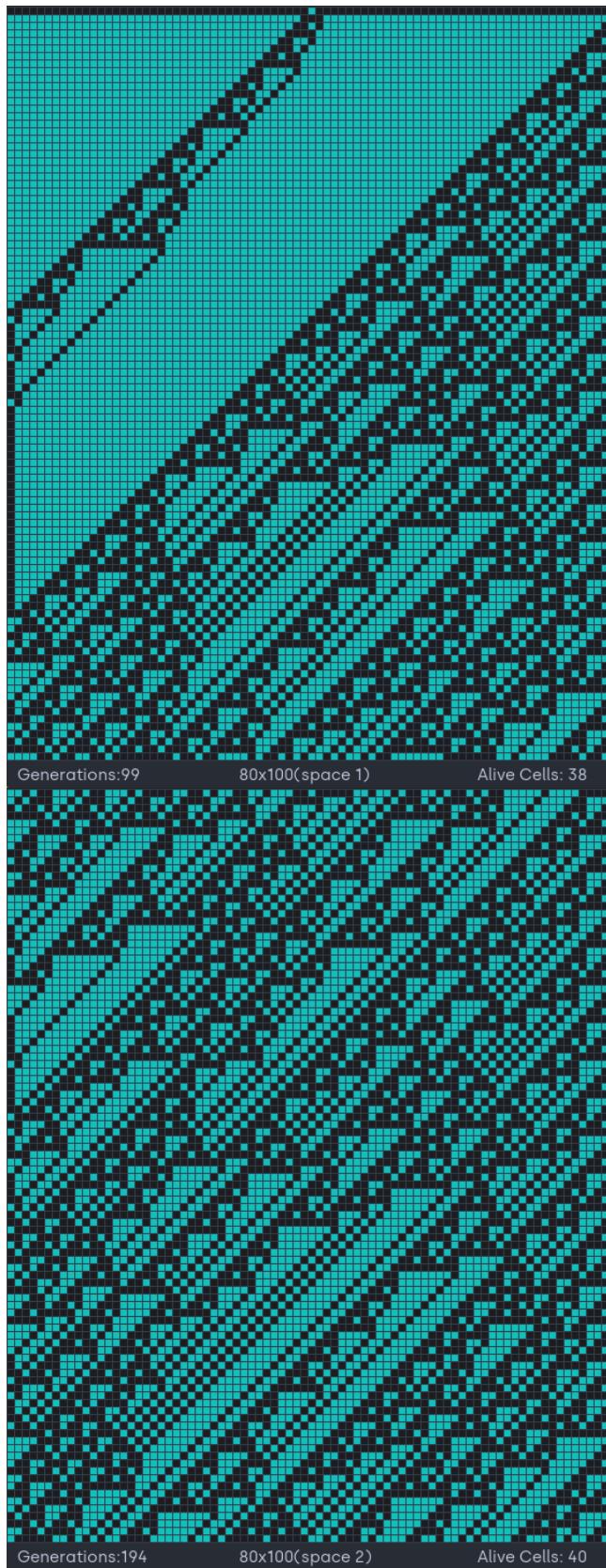


Figure 18: Se muestra de forma continua un par de pantallas resultantes de la evolución de una configuración inicial de célula central con regla 169 durante un total de 99 generaciones en la primer pantalla y finalmente 194 alcanzadas en la segunda.

Notar en la imagen de la segunda pantalla el cambio en el texto central a "space 2", indicando el número de pantalla que se muestra.

2.5 Graficación

Parte de un análisis estadístico de los espacios de evolución, se tiene la graficación de parámetros que ayuden a describir de alguna u otra forma al espacio de evolución, y en más específico la generación actual.

Se habilita en la simulación la graficación de 3 parámetros mediante el uso de 3 botones en su respectiva sección(Figura 5)

2.5.1 Densidad

El primero de los botones disponibles, permite graficar cuando presionado el número de células vivas(densidad) en todas las generaciones evolucionadas hasta el momento.

Su implementación es bastante sencilla. Haciendo uso de una lista, se agregan a esta el número de células nuevas al terminarse el cálculo de la nueva generación del espacio de evolución, y utilizando estos valores se pasan como argumentos para la graficación ocupando la biblioteca matplotlib.

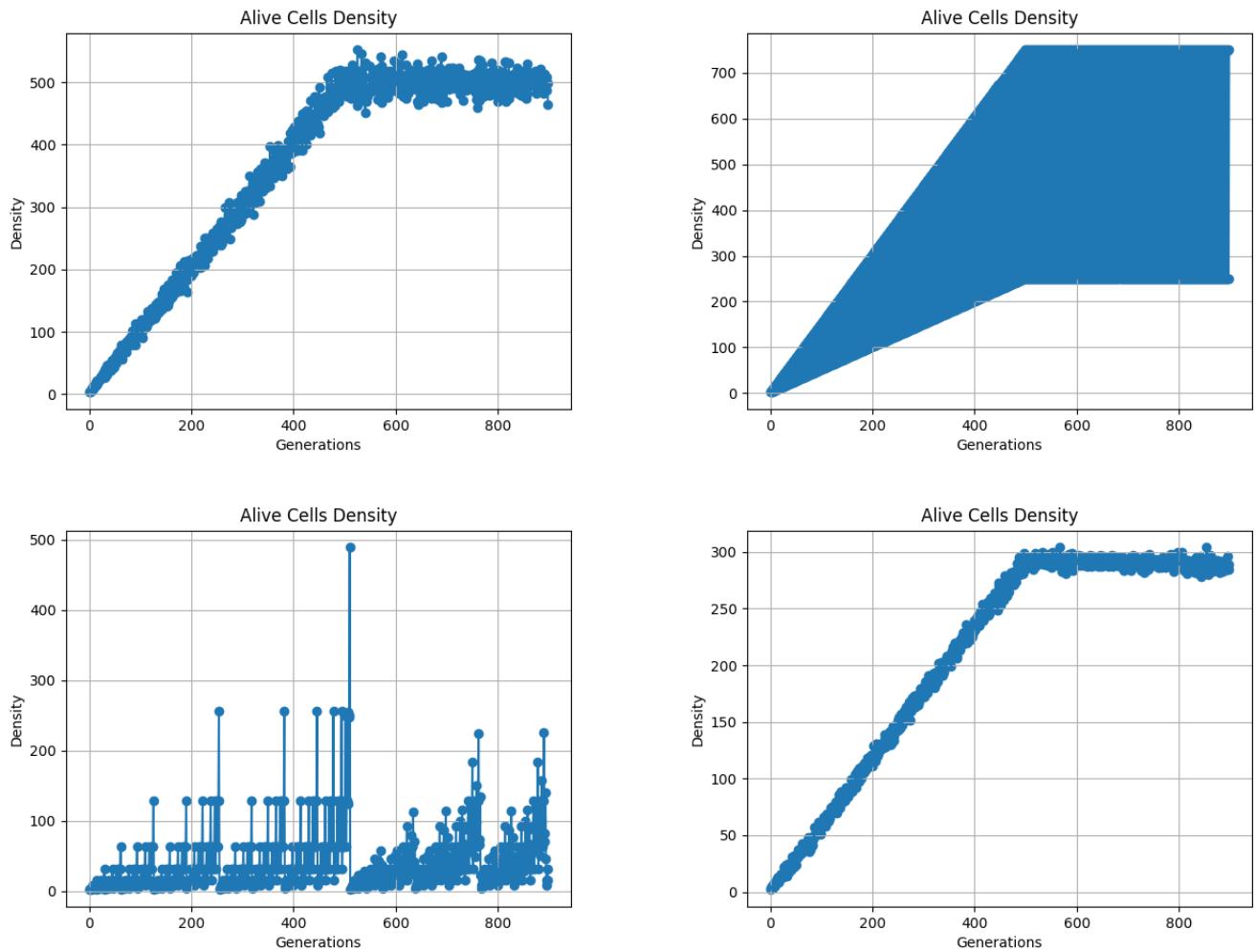


Table 1: Gráficas de densidad después de 900 generaciones para diferentes reglas de evolución en un mismo tamaño de espacio de evolución(1000x900) y con una única célula central como configuración de inicio.

Reglas de evolución aplicadas de izquierda a derecha y arriba hacia abajo: a) 30, b) 54, c) 90, d) 110

2.5.2 Logaritmo Densidad

Mismo funcionamiento y parámetro que la gráfica de densidad, con la única particularidad de graficarse después de aplicarse el logaritmo a la lista de valores: $\log_{10}(Densidad)$

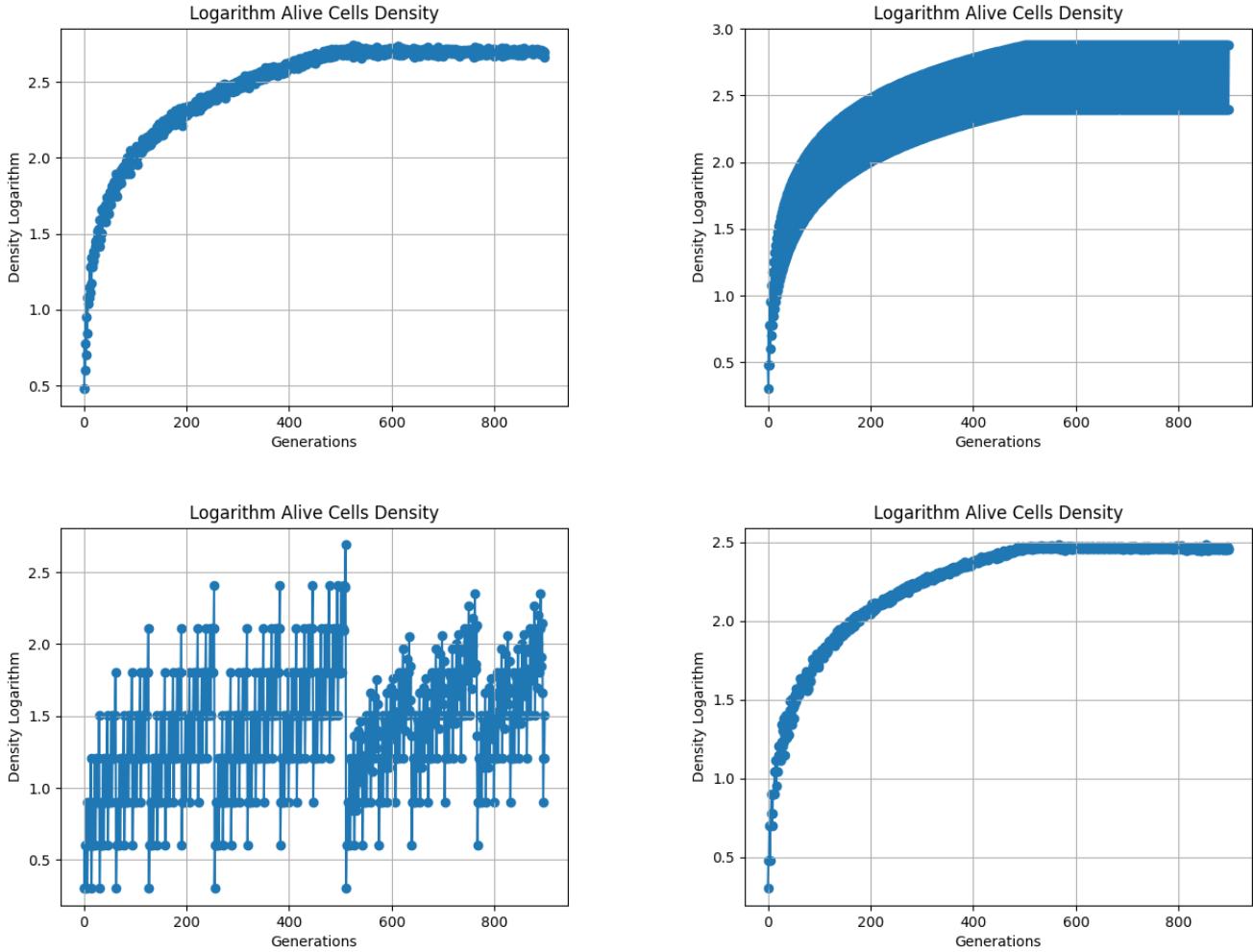


Table 2: Gráficas del logaritmo de la densidad después de 900 generaciones para diferentes reglas de evolución en un mismo tamaño de espacio de evolución(1000x900) y con una única célula central como configuración de inicio.
 Reglas de evolución aplicadas de izquierda a derecha y arriba hacia abajo: a) 30, b) 54, c) 90, d) 110

2.5.3 Entropía

La entropía de Shannon en el ámbito de la teoría de información, se trata de una cantidad que indica la incertidumbre de una fuente de información, también pudiendo interpretarse como la cantidad de información promedio que contienen los símbolos utilizados. Los símbolos con menor probabilidad de aparición serán los que aporten mayor información a la medida, de forma que llegan aumentar o disminuir el valor de esta.

Cuando se tenga una distribución homogénea de los símbolos existentes(cada símbolo diferente tendrá la misma probabilidad), entonces la entropía toma el valor de 1, mientras más símbolos diferentes en distribución heterogénea existan en un conjunto, más alejado el valor de la entropía se encontrará del 1.

Partiendo del predicado ”La entropía mide la cantidad de certidumbre en la incertidumbre”, resulta natural asociar la certidumbre de algo que pase con la probabilidad P , entonces la incertidumbre debe ser el inverso de la Probabilidad $\frac{1}{P}$. La multiplicación de los dos términos satisface el predicado, sin embargo si se implementara de esta forma, obtendríamos que cuando un símbolo tiene probabilidad 1 la medida de incertidumbre sería igualmente 1, pero lo correcto sería que este fuera 0 pues estamos completamente seguros que cualquier símbolo escogido siempre será el mismo, por esta razón se implementa el logaritmo:

$$H = \sum p(x) \log \left(\frac{1}{p(x)} \right)$$

Aplicando propiedades de logaritmos:

$$H = \sum p(x)(\log(1) - \log(p(x))) = \sum (p(x)(0) - p(x)\log(p(x)))$$

Resultando finalmente en la ecuación planteada por Claude Shannon en 1948:

$$H = -\sum p(x)\log(p(x))$$

A partir de esta ecuación lo que se quiere implementar para su graficación en la simulación, es calcular el desorden o diferencias en las vecindades existentes en un espacio de evolución después de calcular su nueva generación. Para esto primero se requiere contabilizar la frecuencia de aparición de cada tipo de vecindad en el espacio, la solución utilizada para esta contabilización fue la de asignar un número único a cada tipo de vecindad, y la forma más sencilla es tomando a los elementos de la vecindad como un número binario, donde naturalmente la posición de cada elemento está ponderado y le corresponde una potencia de 2 para su conversión a decimal. Al convertir esta matriz con sus valores a un valor decimal, este valor sirve como id que identifica al tipo de vecindad permitiendo el incremento de su frecuencia cuando contabilizado.

Después de la contabilización de frecuencias de cada vecindad se realiza la división por el número de elementos en el espacio, obteniendo así la probabilidad de aparición de cada vecindad en esa generación para el espacio de evolución.

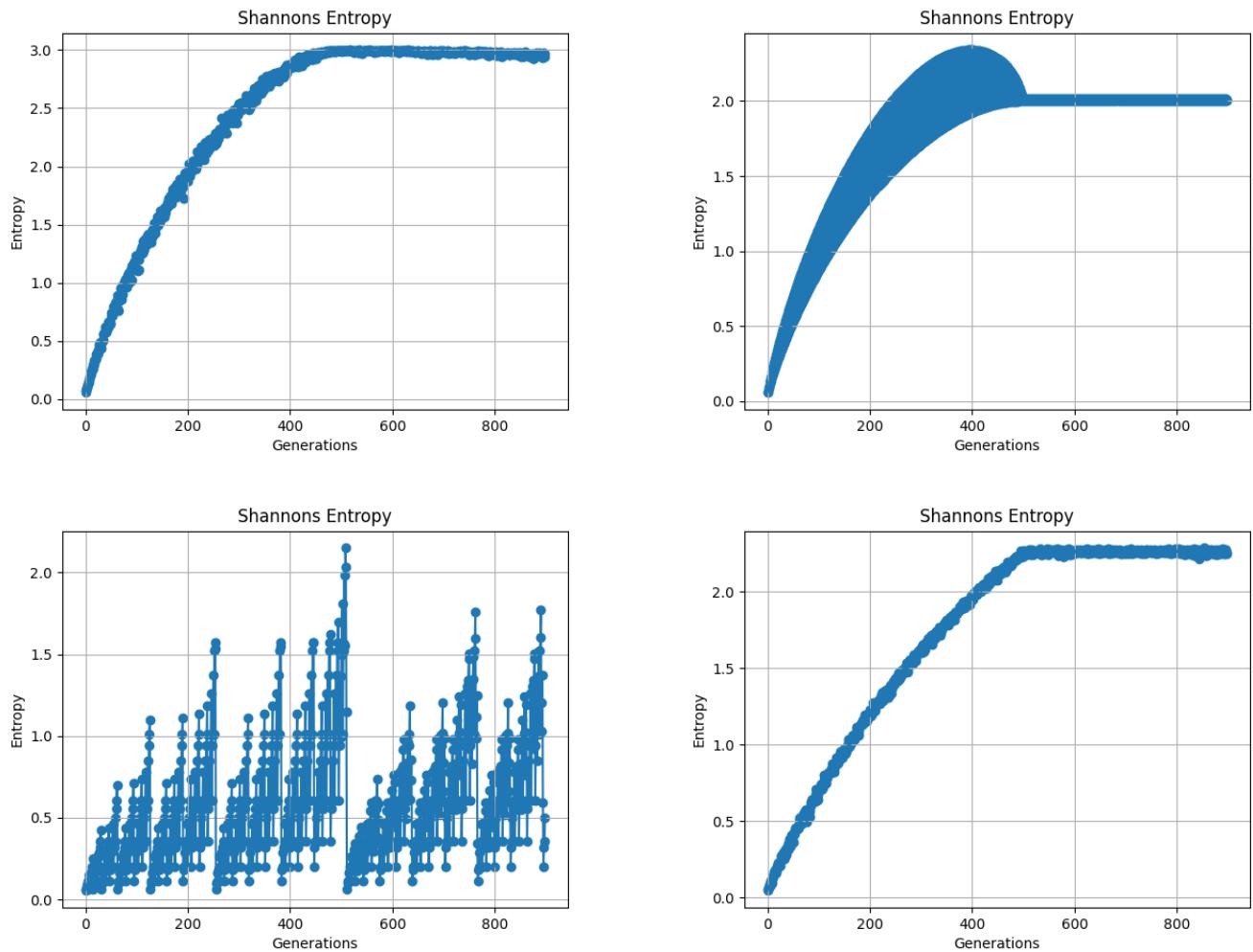


Table 3: Gráficas de la Entropía de Shannon después de 900 generaciones para diferentes reglas de evolución en un mismo tamaño de espacio de evolución(1000x900) y con una única célula central como configuración de inicio.
Reglas de evolución aplicadas de izquierda a derecha y arriba hacia abajo: a) 30, b) 54, c) 90, d) 110

2.6 Cálculo de Attractores

El cálculo de los attractores para una configuración de tamaño n y para una cierta regla de evolución r , requiere del cálculo inicial del universo binario con potencia n , pues la lista resultado de esta operación provee todas las combinaciones de cadenas binarias de tamaño n que serán utilizadas como configuraciones iniciales de evolución con el objetivo de describir el o los árboles de evolución.

Cada cadena binaria corresponderá con una configuración en su forma de células en 1D para su representación en ECA, de forma que para cada universo potencia se tienen en total 2^n hojas o nodos.

Escencialmente se identifican 2 tipos de nodos con importancia particular:

- Nodo raíz: Este nodo refiere a una configuración en el ECA que es imposible de obtener mediante un proceso de evolución con cualquier otro tipo de configuración inicial. Estos nodos son el origen de un árbol propio desde el cual se generan demás nodos, siendo posible la intersección de estos subárboles dando la posibilidad de unirlos formando 1 solo árbol con múltiples nodos raíces.
- Nodo atractor: Para cada árbol generado en un estudio de este tipo, se identifica un nodo atractor. Este es el nodo con más número de predecesores(nodos que al evolucionar lo generan a este) formando parte del ciclo principal de cada árbol. Al final este tipo de nodos es el estado al que tiende la configuración a converger.

Definidos por el usuario, los parámetros principales que son la regla de evolución y la potencia del universo binario, implementado en este caso como un rango de potencias $[n, m]$, el proceso inicia calculando las cadenas binarias de tamaño n hasta m . Con la lista resultante del proceso anterior, se inicia tomando la primer cadena disponible y asignándolo como configuración inicial, para posteriormente realizar de forma consecutiva la evolución del espacio.

La evolución de un espacio en esta rutina tiene como condición de parada la repetición de una configuración previamente calculada, de forma que un árbol se va generando a través de la aplicación de la regla a la configuración inicial hasta que sea identificado que en este propio ciclo o en otro previo, se ha calculado previamente el árbol de evoluciones de la configuración actual, este es el parametro que permite al algoritmo parar las evoluciones consecutivas.

Para lograr esta condición de parada se realiza una copia de la lista del universo binario para cada respectiva potencia que van desde n hasta m . Tomando desde la configuración más pequeña como configuraciones de inicio, cuando se obtenga una configuración nueva a través de la evolución, se indica en este arreglo copia modificando el valor del elemento en el índice correspondiente a la cadena por un valor constante diferente que sirva como bandera para indicar su previo cálculo. En la implementación de una lista de cadenas binarias, cuando se identifica el cálculo de una nueva configuración esta cadena cambia su valor a la bandera ***False***.

El indicar que configuraciones del universo binario han sido ya calculados permite al algoritmo ser más eficiente, evitando colocar como condición inicial a una cadena ya identificada durante un proceso de evolución.

La identificación de los atractores puede realizarse a través de un análisis topológico, estudiando las relaciones entre los nodos resultantes; O de forma gráfica observando los árboles de evolución. Un grafo dirigido puede definirse utilizando variedad de bibliotecas en Python como un conjunto de relaciones binarias entre nodos, exigiendo del algoritmo que durante el proceso de evolución se guarde en una lista de relaciones y en forma de tupla, la descendencia de las configuraciones obtenidas después de 1 evolución.

En el caso particular del análisis topológico, se identifica para cada grafo los ciclos existentes(1 para cada árbol generado), y de los nodos que conforman estos ciclos se identifica como atractor aquel con el mayor número de ancestros, o nodos que utilizando el proceso de evolución resultan en esta configuración. Es posible crear un algoritmo de forma sencilla utilizando las herramientas que provee la biblioteca ***networkx*** para el análisis de grafos.

Para la impresión gráfica basta con solo definir el grafo en función de sus relaciones y con ayuda de la biblioteca ***igraph***, ejecutar la creación y acomodo automático de los nodos para la conformación de una imagen. En el caso particular de esta biblioteca el archivo resultante otorga un EPS(Encapsulated PostScript).

La impresión gráfica también se vale de una ayuda del análisis topológico, permitiendo colorear los estados identificados como atractores de un color amarillo que permite contrastar fácilmente entre los demás nodos. Esta ayuda visual se vuelve primordial a medida que se comparan los grafos resultantes cuando se aumenta el valor de n como potencia del universo binario. Al punto que en esta simulación el resultado topológico es el único capaz de interpretar pues son tantos

el número de nodos en la imagen que es imposible para la biblioteca realizar un acomodo ordenado dentro del espacio de impresión para ser identificable. Hay que recordar que el crecimiento del número total de nodos entre uno y otro universo es de orden exponencial.

Finalmente durante el proceso de cálculo, y fungiendo al mismo tiempo como indicador de la ejecución del proceso y descriptor gráfico de los resultados progresivos obtenidos del algoritmo, en el espacio de evolución se imprimen las configuraciones siendo evaluadas así como sus predecesores que integran los árboles de evolución resultantes.

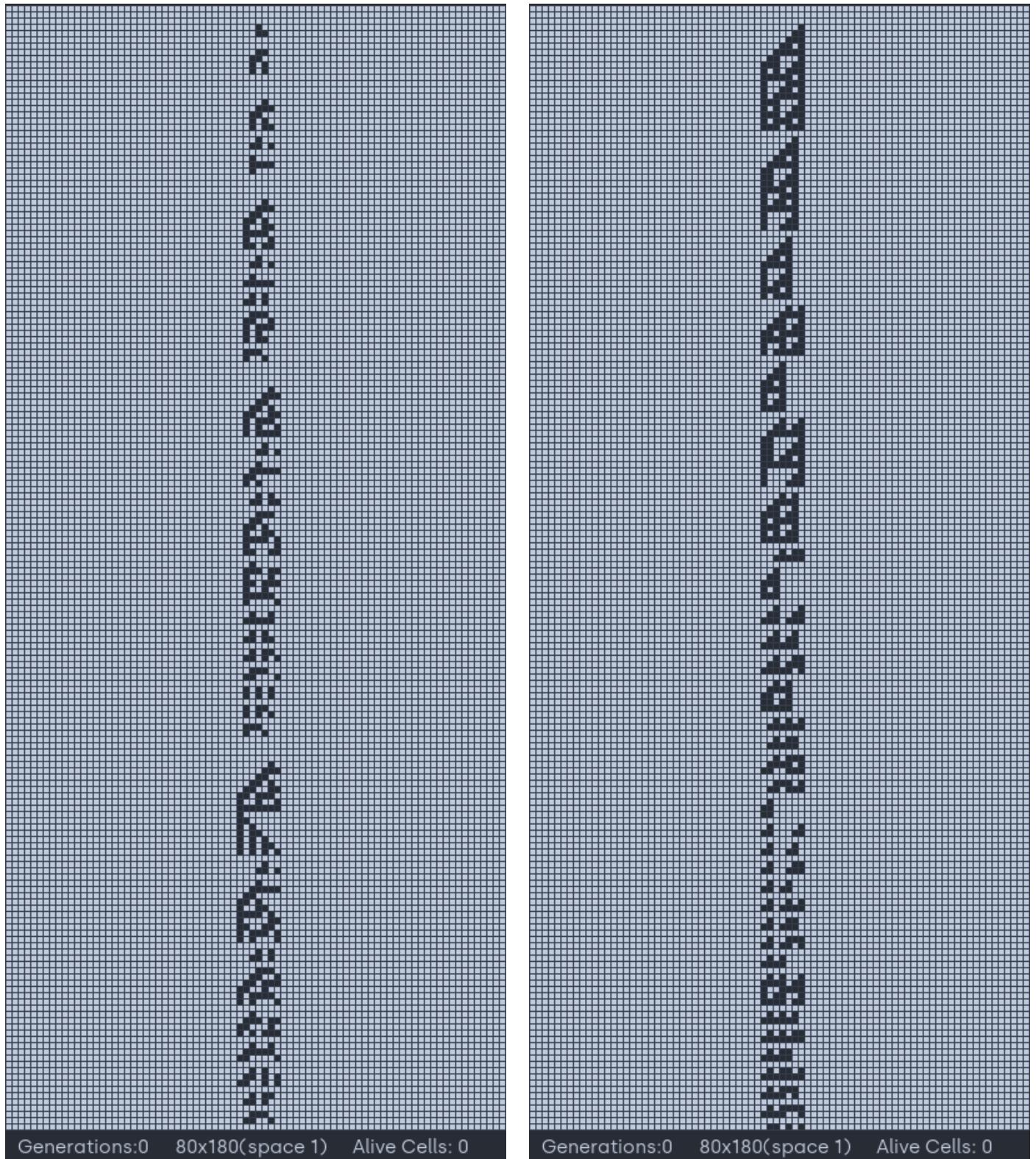


Table 4: Resultado gráfico mostrado durante el proceso de cálculo de los atractores en las reglas 30 y 110 respectivamente

3 Análisis de atractores

La ciencia sostiene que existe una capacidad espontánea de las cosas para organizarse, siendo una propiedad inherente de los sistemas complejos. En este tipo de sistemas donde se poseen múltiples componentes que se relacionan de múltiples maneras y resultan múltiples trayectorias derivadas de su interacción, existen coincidencias o choques en el espacio que harán aparecer anomalías, cosas nuevas derivadas de circunstancias impredecibles que constituyen una modificación potencialmente organizadora en un sistema. A esta condición emergente se le denomina **Atractor**.

La idea de los atractores se desarrolla en la teoría de los sistemas dinámicos y complejos, derivado del problema de la física clásica "*Problema de los 3 cuerpos*". Si la cantidad de posibilidades para predecir el comportamiento gravitacional de tres masas correlacionadas incluye soluciones caóticas (la línea histórica Newton/Laplace/Poincaré/Lorenz), la de miles de millones de elementos de cualquier sistema es aún mucho más "infinita".

Los atractores pueden ser un conjunto de valores numéricos hacia los que un sistema tiende a evolucionar, geometricamente pueden tomar la apariencia de un punto, una curva, una variedad o incluso un conjunto complicado de estructuras fractales, y sin embargo para que un conjunto sea considerado un atractor debe cumplir que las trayectorias que le sean suficientemente próximas han de permanecer próximas incluso si son ligeramente perturbadas, por lo que no propiedad especial debe ser satisfecha excepto la de que la trayectoria del sistema permanezca en el atractor.

Esta misma condición puede ser estudiada en la evolución de los ECAs en sus diferentes reglas, pues los patrones generados de la aplicación de las diferentes reglas describen un comportamiento distinto entre ellas.

Así en el contexto de los ECAs como un sistema dinámico que evoluciona condicionado por la regla de evolución implementada, los estados de evolución tienen una tendencia a estabilizarse en una estructura en específico. Esta estructura corresponde a un atractor y el objetivo de este corto estudio es encontrar la estructura o estado atractor de una regla.

Para algunas de las reglas existe una dinámica de atractores múltiples dentro el sistema, lo que describiría que ese sistema evolucionando con la regla en específico tiene una alta tendencia a estabilizarse sin la necesidad de evolucionar durante muchas generaciones. Debido a esto difícilmente será posible anotar específicamente el número de atractores existentes para tamaños de universos binarios potencia muy grandes, por lo que se en cambio se reportan las observaciones sobre patrones o tendencias identificadas en estos casos.

3.1 Análisis de los campos de atracción

La meta del estudio consiste en encontrar y analizar los campos de atracción de los ECA cuando se implementan las 256 reglas de evolución posibles, y en un rango de valores para la potencia del universo binario(longitud de las configuraciones a evolucionar). Afortunadamente para este estudio se han encontrado equivalencias entre ciertas reglas pues muestran propiedades simétricas de reflexión como si la evolución se observara desde un espejo o se rotara sobre el eje de las ordenadas. Otras muestran un patrón completamente igual pero negado (las distribución de las células vivas en una regla muestran la misma evolución pero con aquellas que están muertas), y otras muestran las 2 propiedades al mismo tiempo, de forma que la lista de 256 reglas se acorta a tan solo 88 y se va a considerar el valor de n desde 0 hasta 15.

Regla	Equivalentes	Regla	Equivalentes
0	255	56	98, 185, 227
1	127	57	99
2	16, 191, 247	58	114, 163, 177
3	17, 63, 119	60	102, 153, 195
4	223	62	118, 131, 145
5	95	72	237
6	20, 159, 215	73	109
7	21, 31, 87	74	88, 173, 229
8	64, 239, 253	76	205
9	65, 111, 125	77	-
10	80, 175, 245	78	92, 141, 197
11	47, 81, 117	90	165
12	68, 207, 221	94	133
13	69, 79, 93	104	233
14	84, 143, 213	105	-
15	85	106	120, 169, 225
18	183	108	201
19	55	110	124, 137, 193
22	151	122	161
23	-	126	129
24	66, 189, 231	128	254
25	61, 67, 103	130	144, 190, 246
26	82, 167, 181	132	222
27	39, 53, 83	134	148, 158, 214
28	70, 157, 199	136	192, 238, 252
29	71	138	174, 208, 224
30	86, 135, 149	140	196, 206, 220
32	251	142	212
33	123	146	182
34	48, 187, 243	150	-
35	49, 59, 115	152	188, 194, 230
36	219	154	166, 180, 210
37	91	156	198
38	52, 155, 211	160	250
40	96, 235, 249	162	176, 186, 242
41	97, 107, 121	164	218
42	112, 171, 241	168	224, 234, 248
43	113	170	240
44	100, 203, 217	172	202, 216, 228
45	75, 89, 101	178	-
46	116, 139, 209	184	226
50	179	200	236
51	-	204	-
54	147	232	-

Table 5: Tabla de las reglas equivalentes para los ECAs

3.1.1 Regla 0

Regla trivial donde todos los estados en una etapa de evolución se estabilizan en el estado **0**, el único y principal atractor de esta regla.

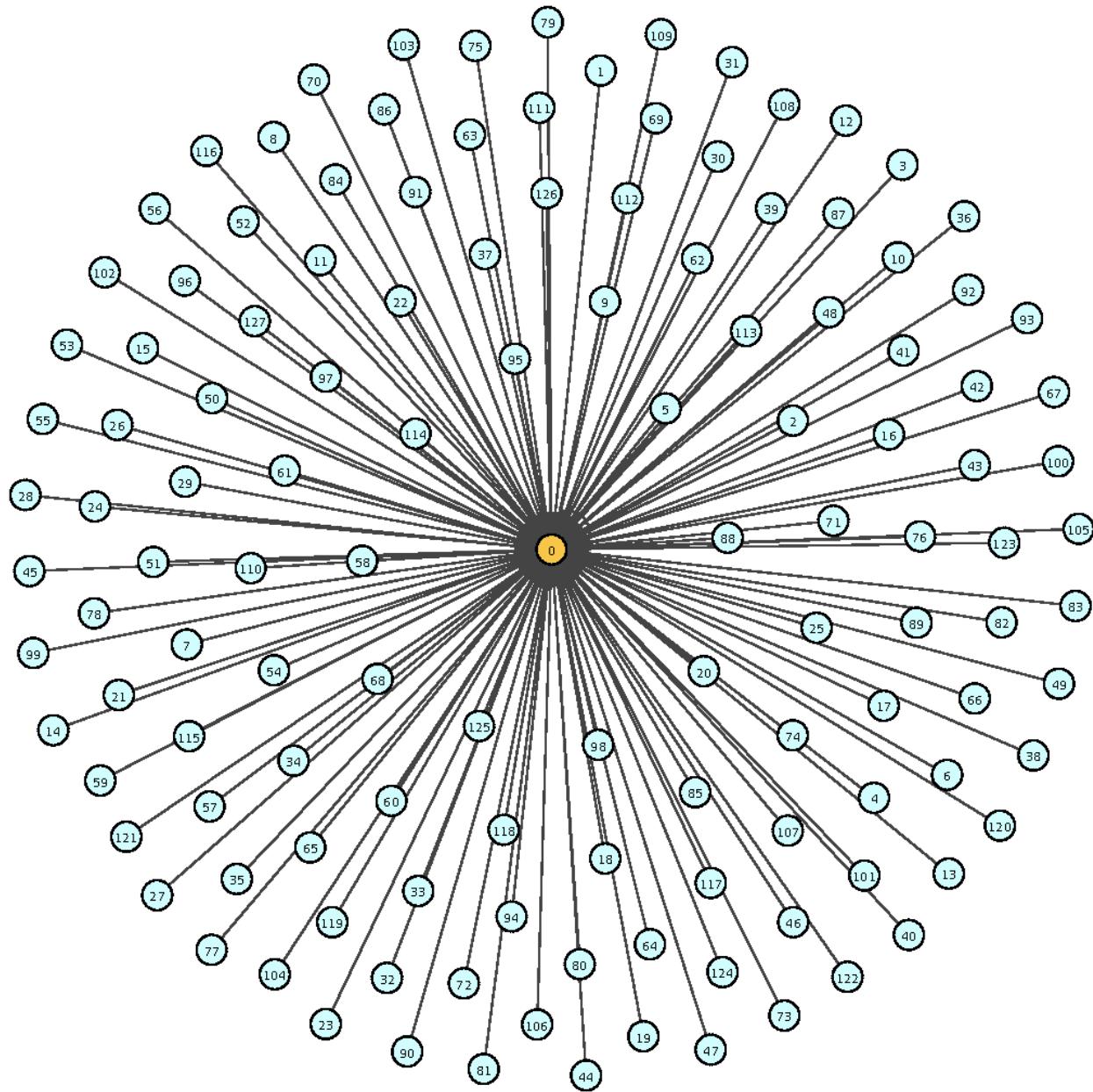


Figure 19: Grafo del atractor 0 para la regla 0 con $n = 5$

3.1.2 Regla 1

Regla sencilla que converge rápidamente a un atractor. El mayor porcentaje de veces con 1 sola etapa de evolución la configuración converge, y aún así existen convergencias con máximo 2 etapas de evolución.

Para valores pequeños de n el número de atractores se mantiene únicamente con 1 al estado 0, sin embargo a medida que n va creciendo el número de atractores diferentes de 0 van apareciendo, llegando a ser 609 atractores para $n = 15$ y más sin embargo siempre el 0 se muestra como un atractor para un gran número de estados.

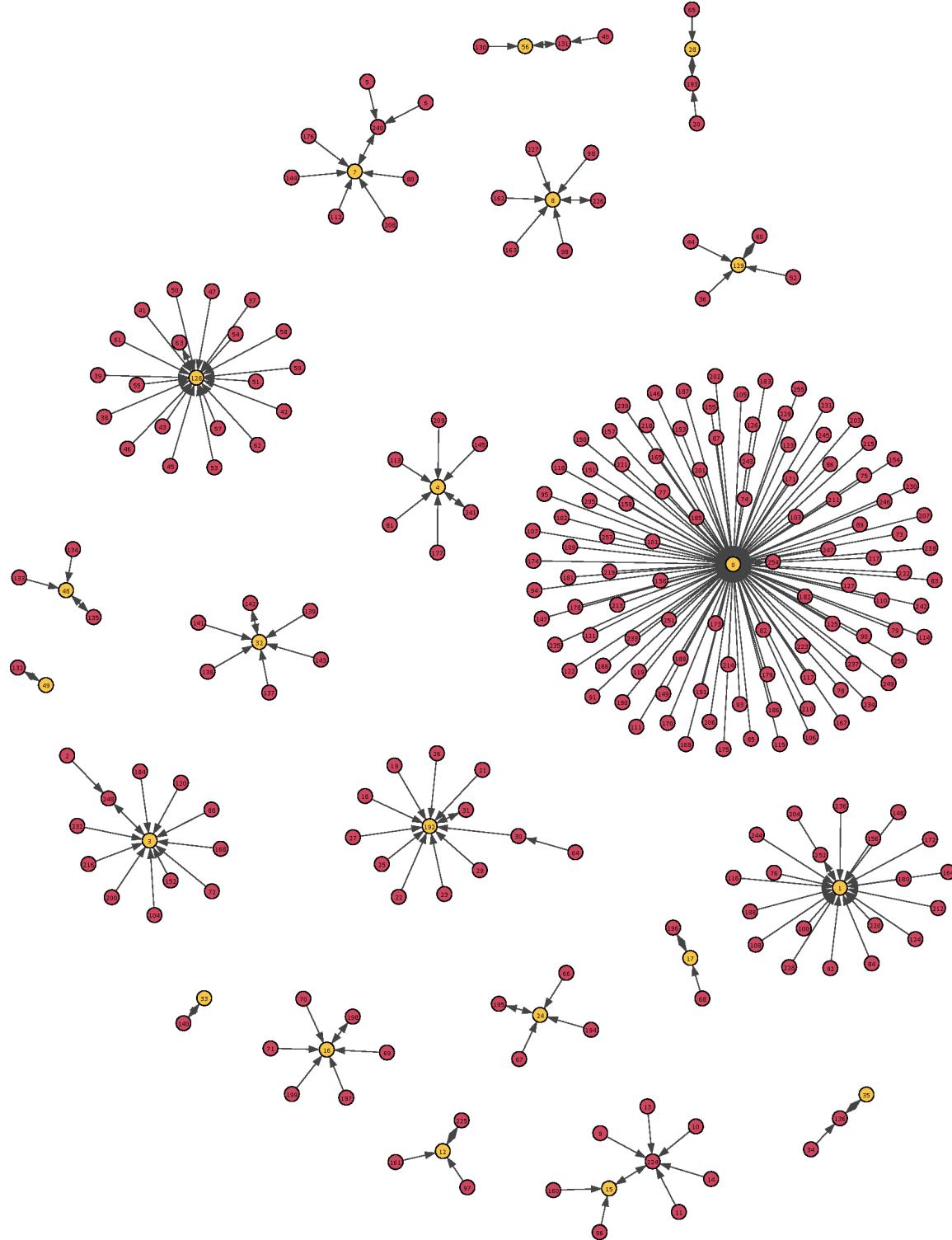


Figure 20: Grafo de los multiples atractores para la regla 1 con $n = 8$

3.1.3 Regla 2

Regla interesante que cuenta con una gran cantidad de nodos raíz o también llamadas hojas del edén, estos estados no pueden ser replicados por el ECA y tan solo existen y evolucionan pues han sido implementados desde el universo binario.

La convergencia de la regla para cualquier tamaño de universo binario será al atractor 0, y sin embargo a medida que aumenta el universo, más fases de evolución le toma al ECA para converger. En el árbol se van definiendo por las ramas de evolución nodos también muy particulares que tienen la propiedad de atracción pues un gran número de nodos raíz como ancestros directos, inician su camino de convergencia a este nodo con 1 sola fase de evolución.

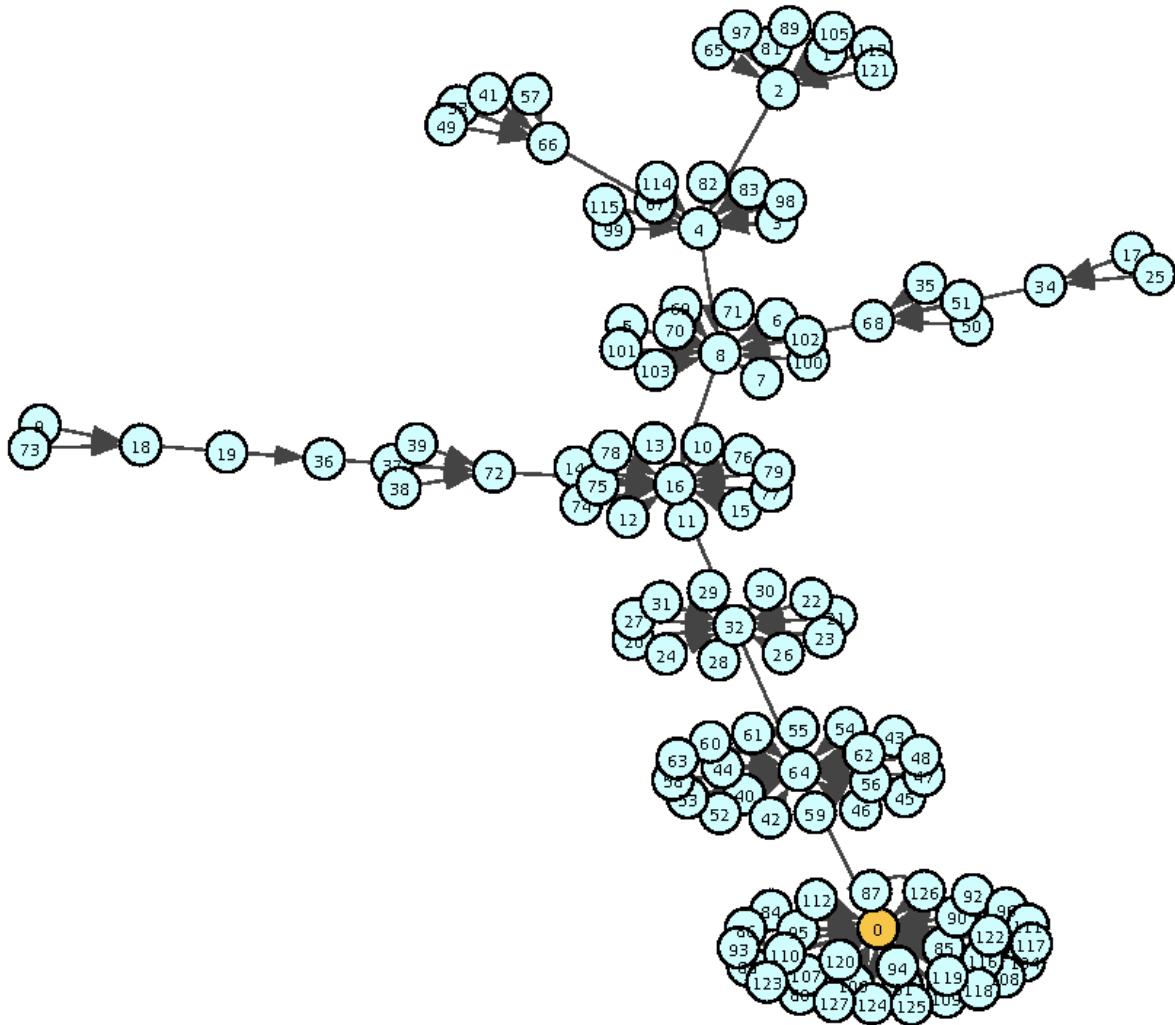


Figure 21: Grafo del atractor 0 único para la regla 2 con $n = 7$

3.1.4 Regla 3

Con un comportamiento similar a la regla 2, esta regla converge al atractor de estado **0** sin importar el tamaño de n , y sin embargo se resalta la diferencia con la regla de 2 de la carencia de la gran cantidad de nodos con propiedades atractoras en el árbol de evolución, lo que indica que el número de evoluciones requeridas para converger al estado 0 se incrementa para esta regla.

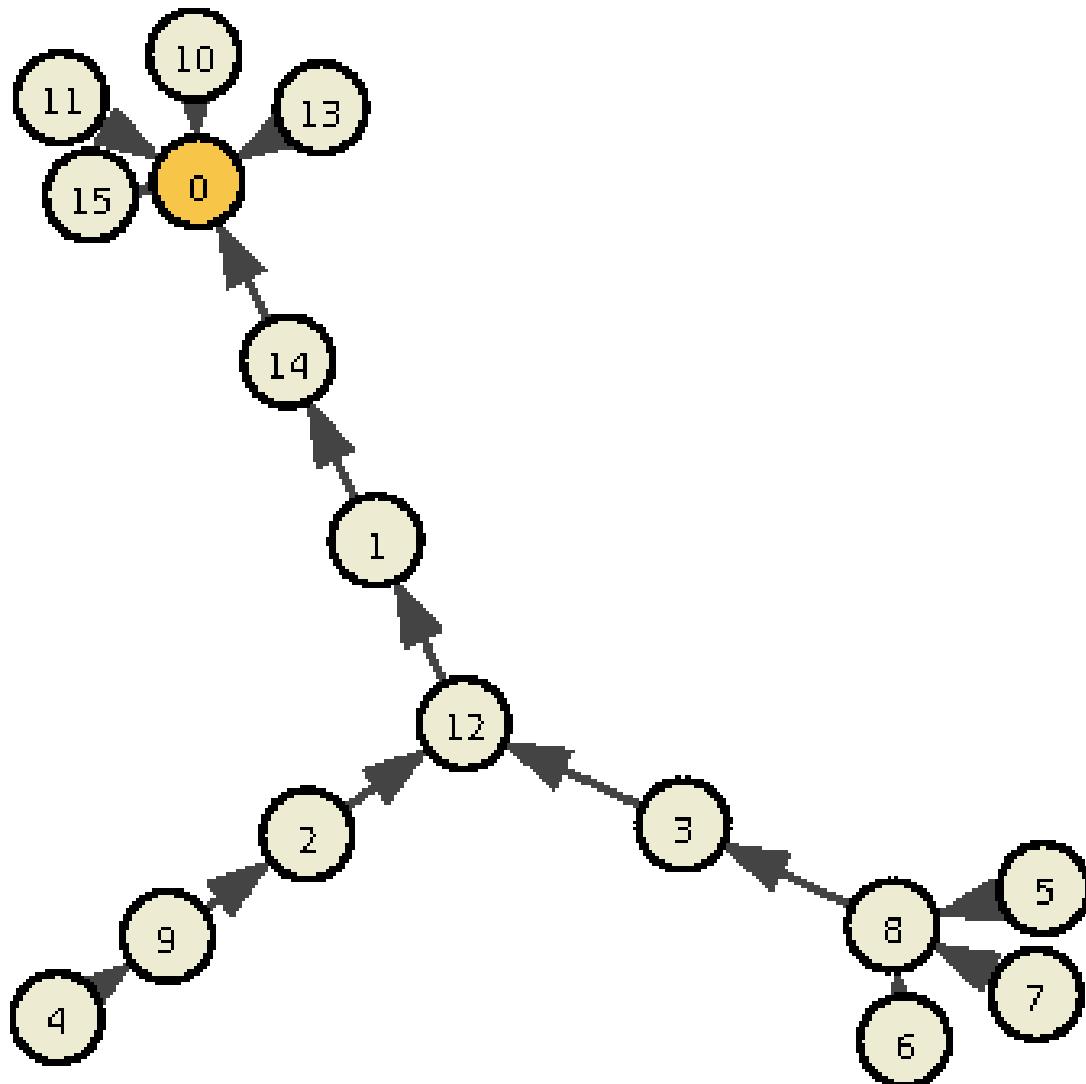


Figure 22: Grafo del atractor 0 único para la regla 3 con $n = 4$

3.1.5 Regla 4

Regla con una rapidez de convergencia de tan solo 1 evolución para todos los estados, aunque no a un solo atractor, pues se definen varios atractores más allá de que el estado **0** se mantienen como el atractor dominante por el número de ancestros que evolucionan a este.

También se observan por primera vez estados que son al mismo tiempo hojas del Edén y atractores de ellos mismos. Estos estados no tienen influencia alguna en el sistema pues no interactúan con ninguna otra configuración y se estabilizan con ellos mismos. Evidentemente este tipo de estados nunca podrán ser generados por el ECA, y sin embargo su inexistencia en el sistema no afecta de forma alguna.

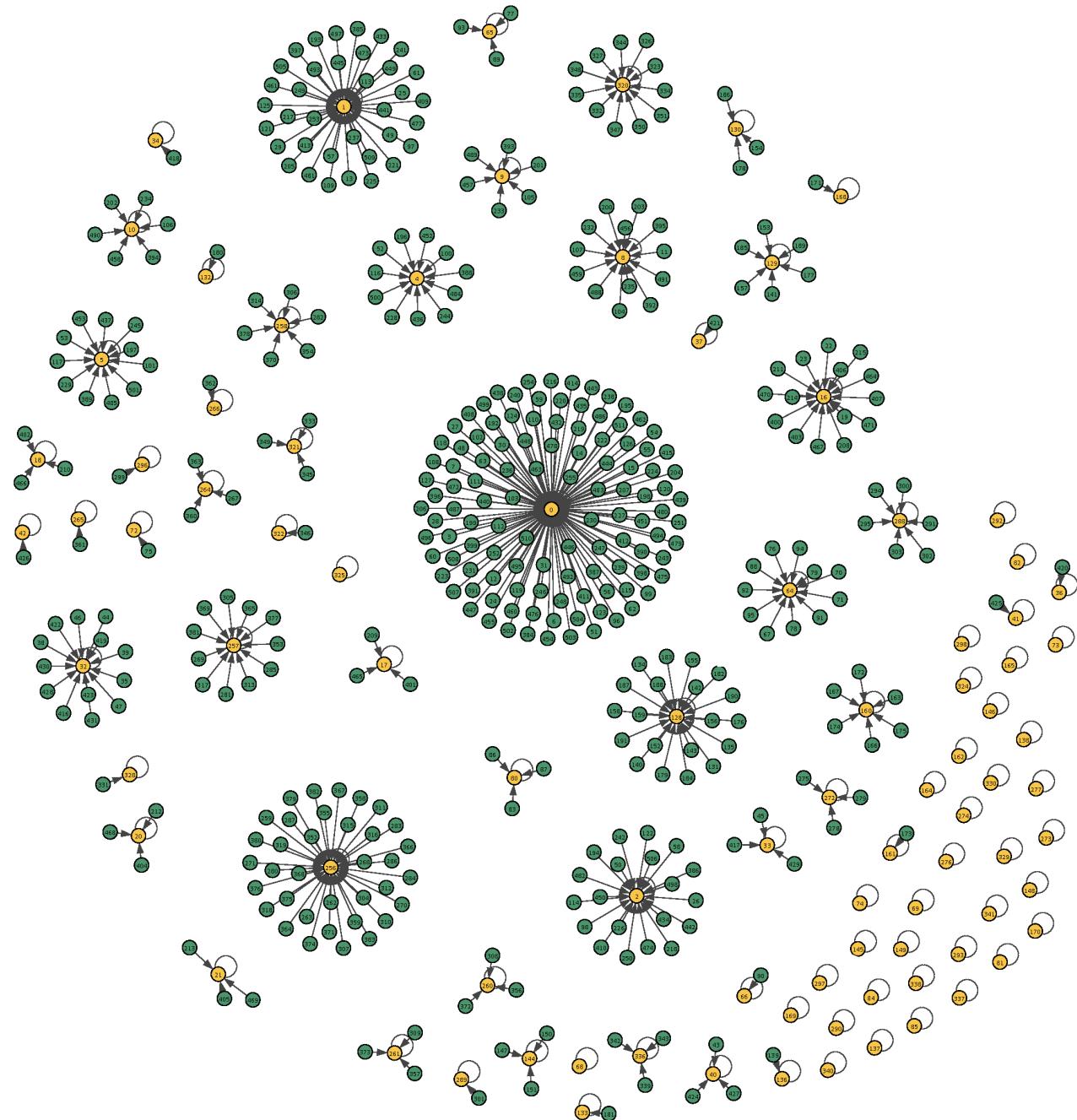


Figure 23: Grafo de los múltiples atractores para la regla 4 con $n = 9$

3.1.6 Regla 5

El comportamiento de los campos de atracción de la regla 5 es sumamente similar a la 4, con algunas diferencias como el número más reducido de nodos atractores marginales(hojas del Edén y atractores al mismo tiempo), así como una mayor extensión de arboles con más de 1 fase de evolución para la convergencia.

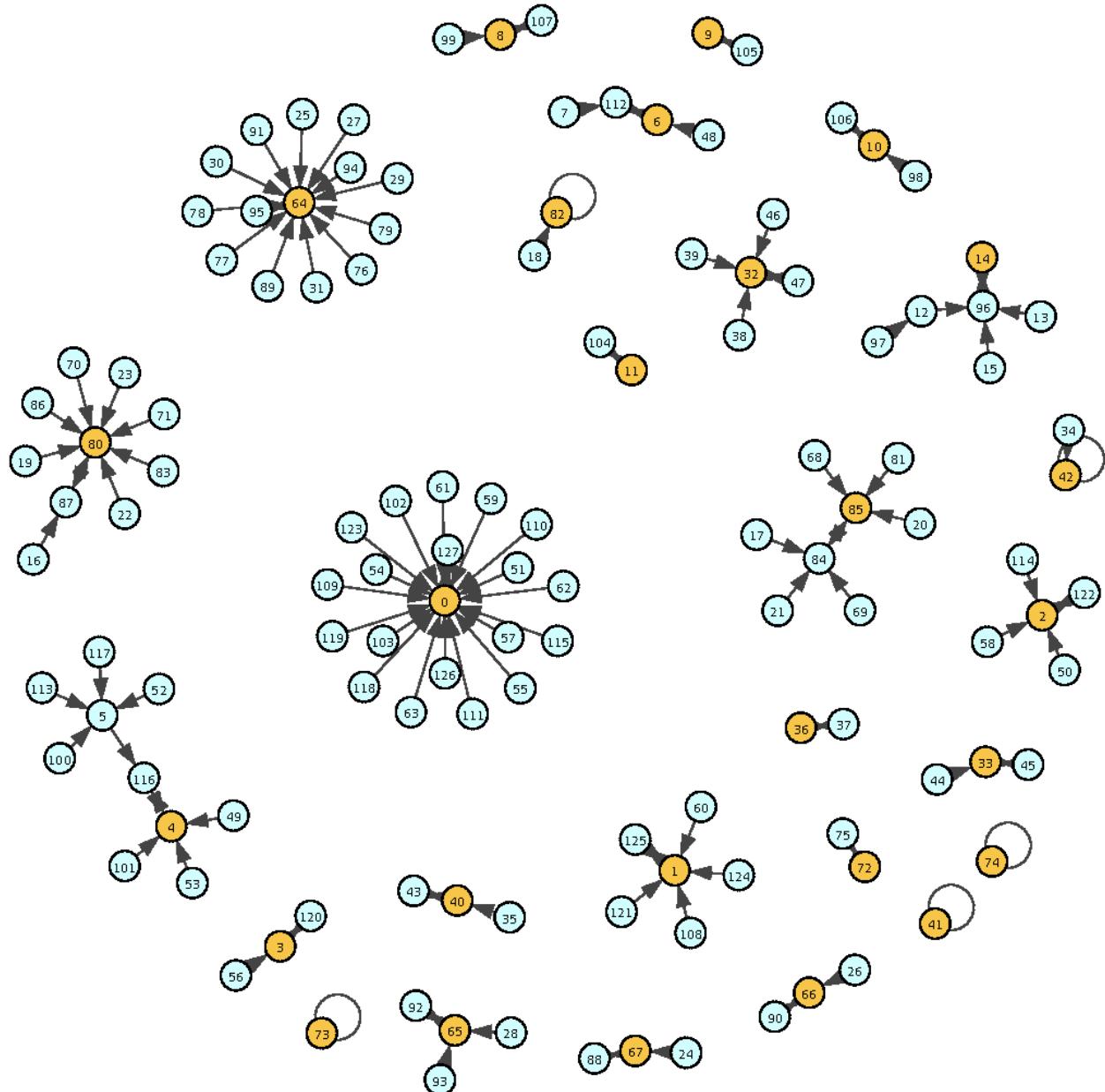


Figure 24: Grafo de los múltiples atractores para la regla 5 con $n = 7$

3.1.7 Regla 6

En esta regla se desarrollan árboles de evolución con más fases de evolución para lograr la convergencia, aumentando el número de árboles a medida que se aumenta el tamaño de n . Se nota curiosamente que existe para cada una de los valores de n un único atractor marginal, cambiando de estado en cada generación, pero manteniéndose constante en número.

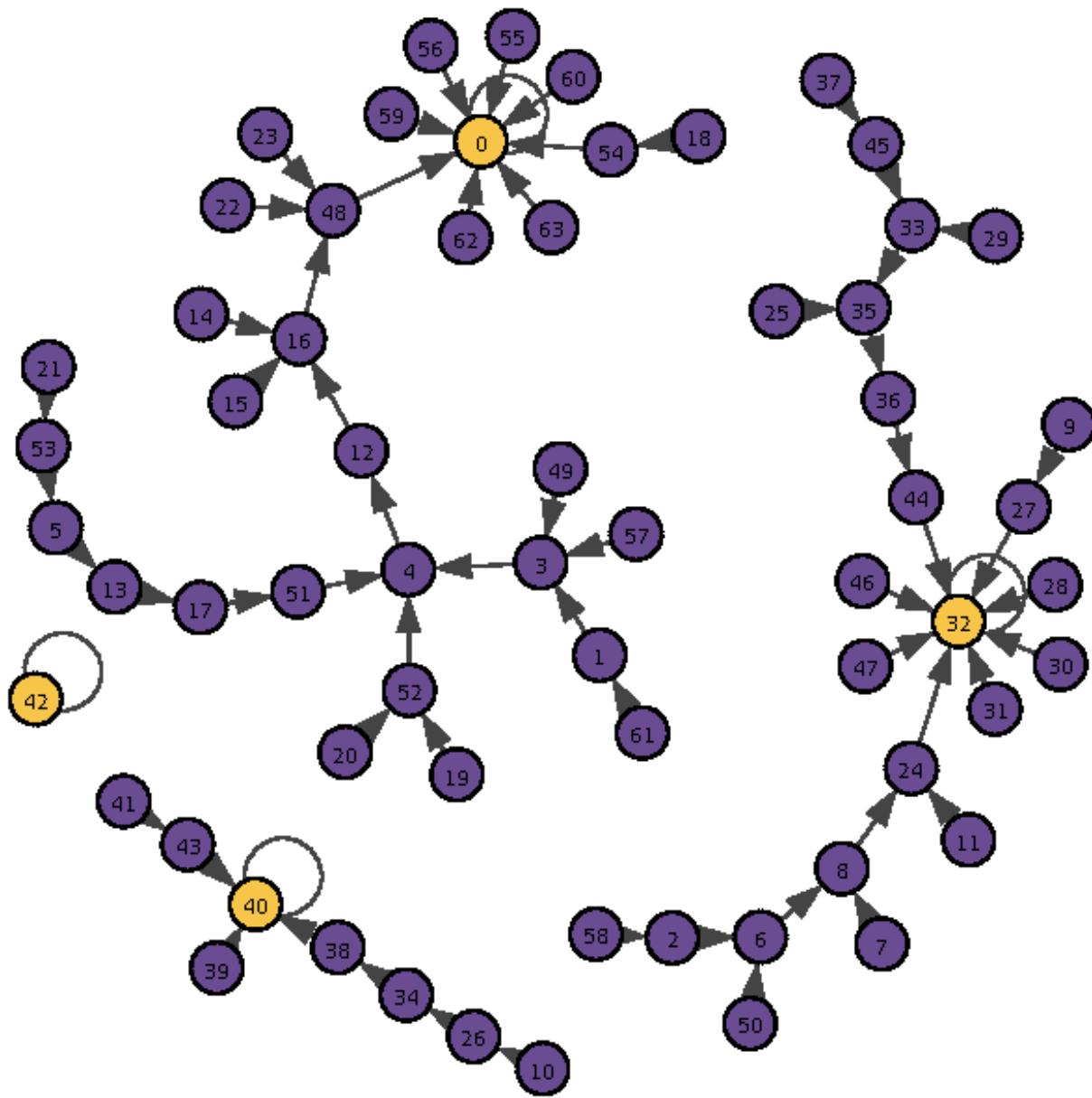


Figure 25: Grafo de los múltiples atractores para la regla 6 con $n = 6$

3.1.8 Regla 7

Regla dominada por árboles de evolución con un gran número de nodos, por lo que la rapidez de convergencia para esta regla es menor. No se observa una mayoría significativa de nodos convergiendo en 1 sola evolución o atractores marginales.

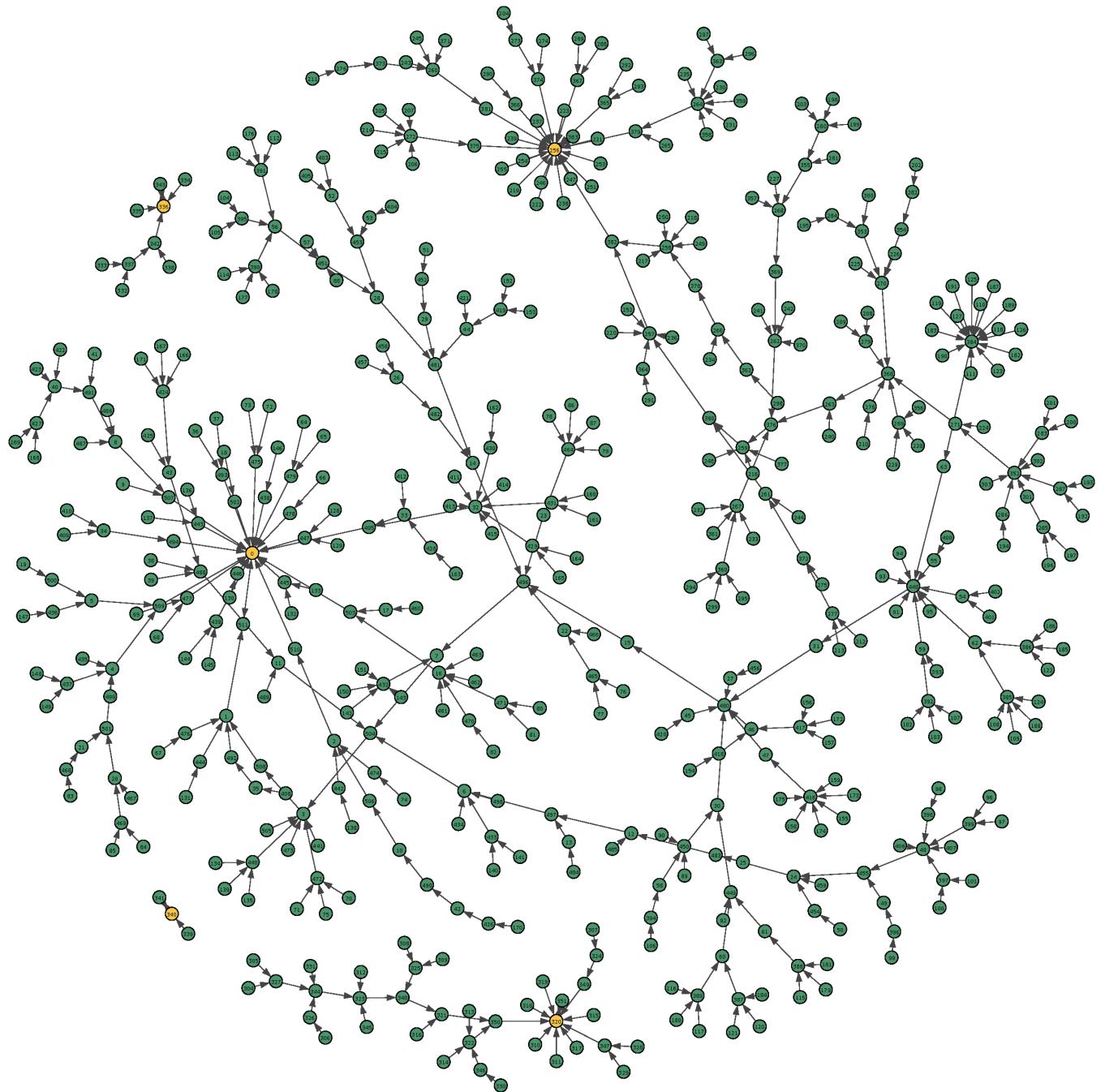


Figure 26: Grafo de los múltiples atractores para la regla 7 con $n = 9$

3.1.9 Regla 8

Con un comportamiento muy similar a la regla 2, la regla converge únicamente al atractor de estado 0, aunque rescatablemente se observan más nodos no atractores con propiedades de atracción, así como una disminuida distancia entre cualquier nodo y la convergencia, por lo que el número de evoluciones necesarias para su estabilización se mantienen baja.

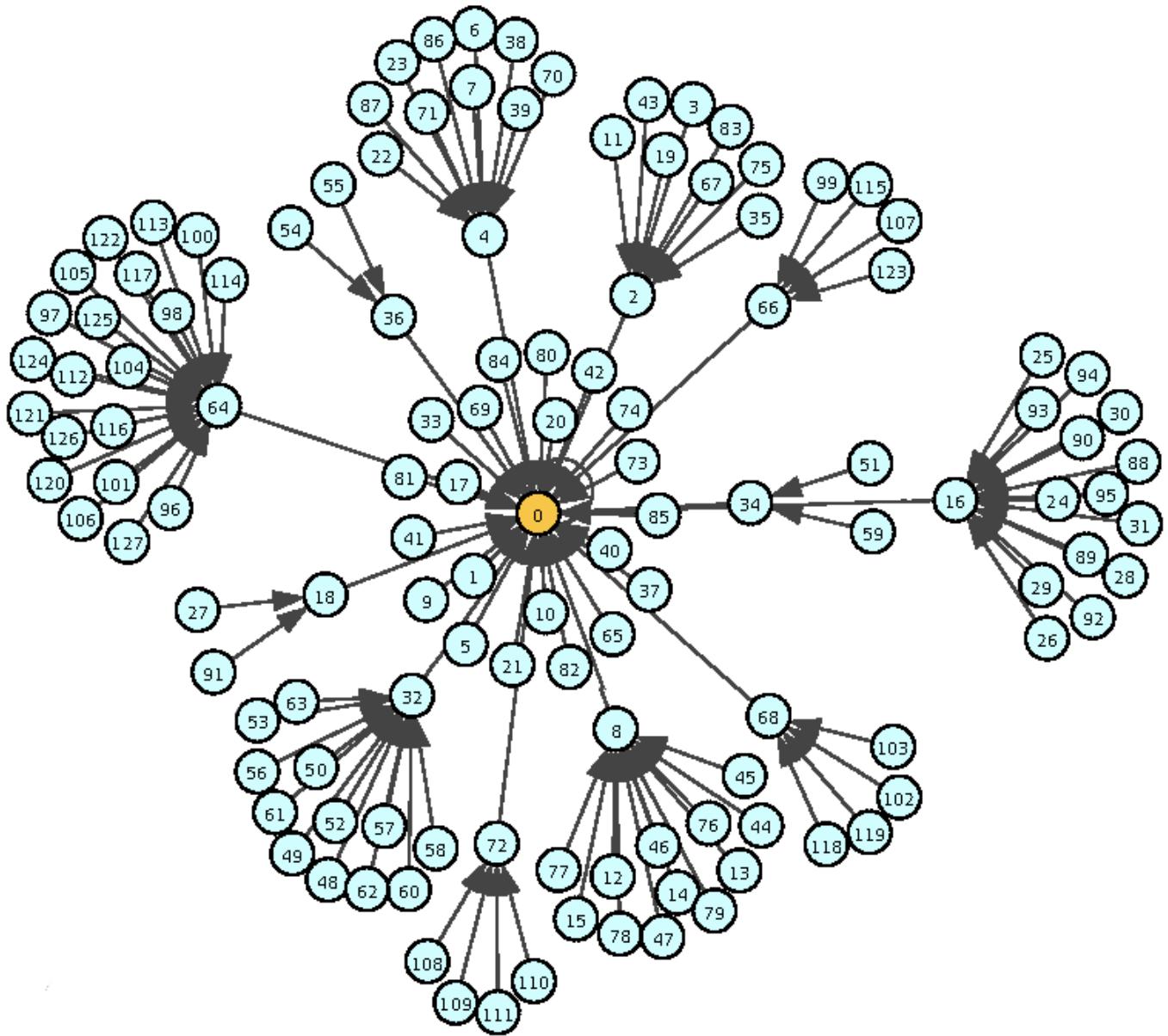


Figure 27: Grafo de del atractor 0 para la regla 8 con $n = 7$

3.1.10 Regla 9

Regla con generación de árboles de evolución extensos pero escasos, por lo que para valores de n pequeños el número de atractores se mantiene entre 1 y 3, alcanzando valores de 9 aumenta este número hasta alcanzar 6 árboles atractores con $n = 14, 15$.

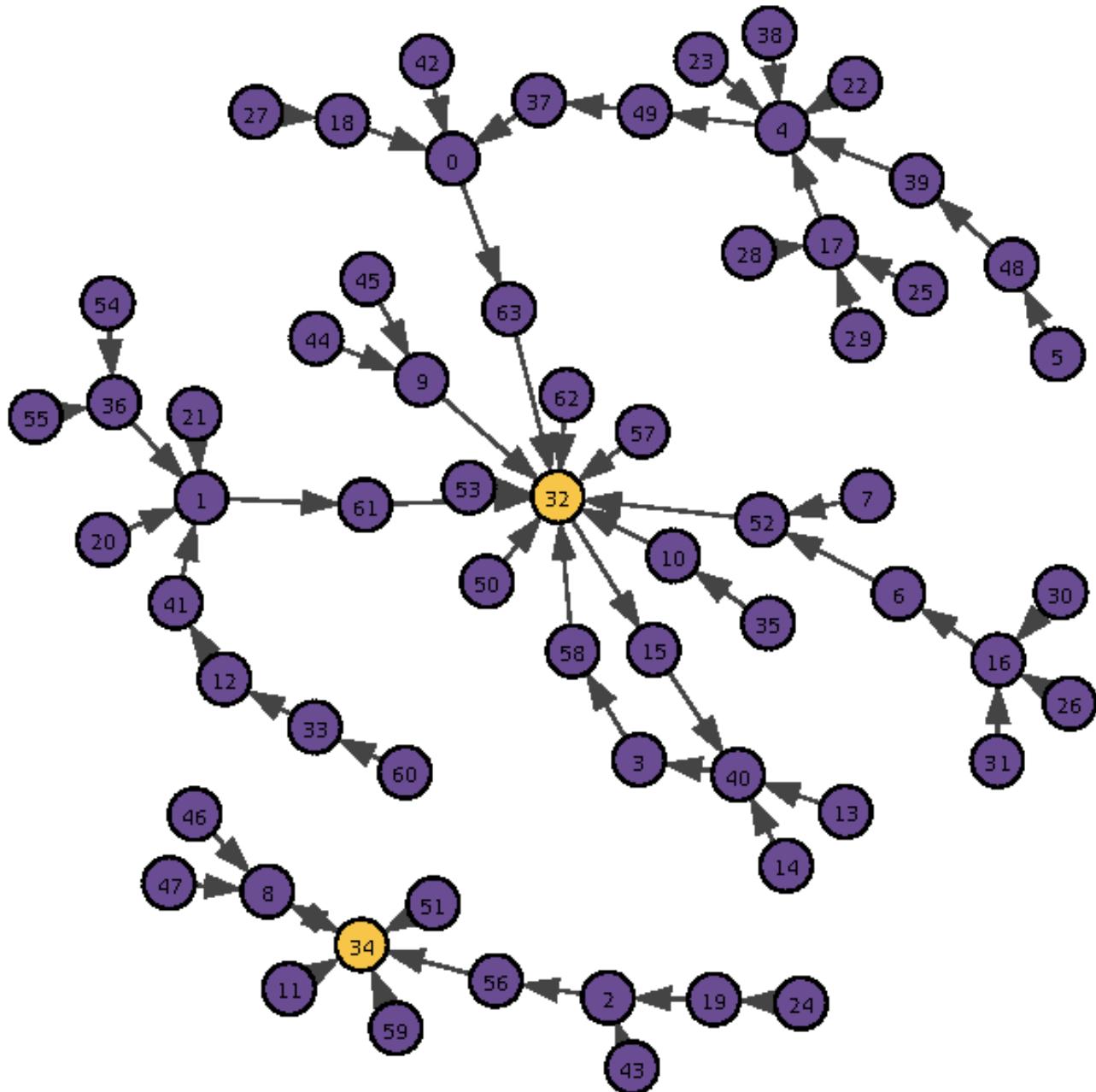


Figure 28: Grafo de los múltiples atractores para la regla 9 con $n = 6$

3.1.11 Regla 10

Comportamiento parecido al de la regla 3 con algunos nodos con propiedades atractoras pero con un árbol de evolución más extendido y nodo atractor general en el estado 0.

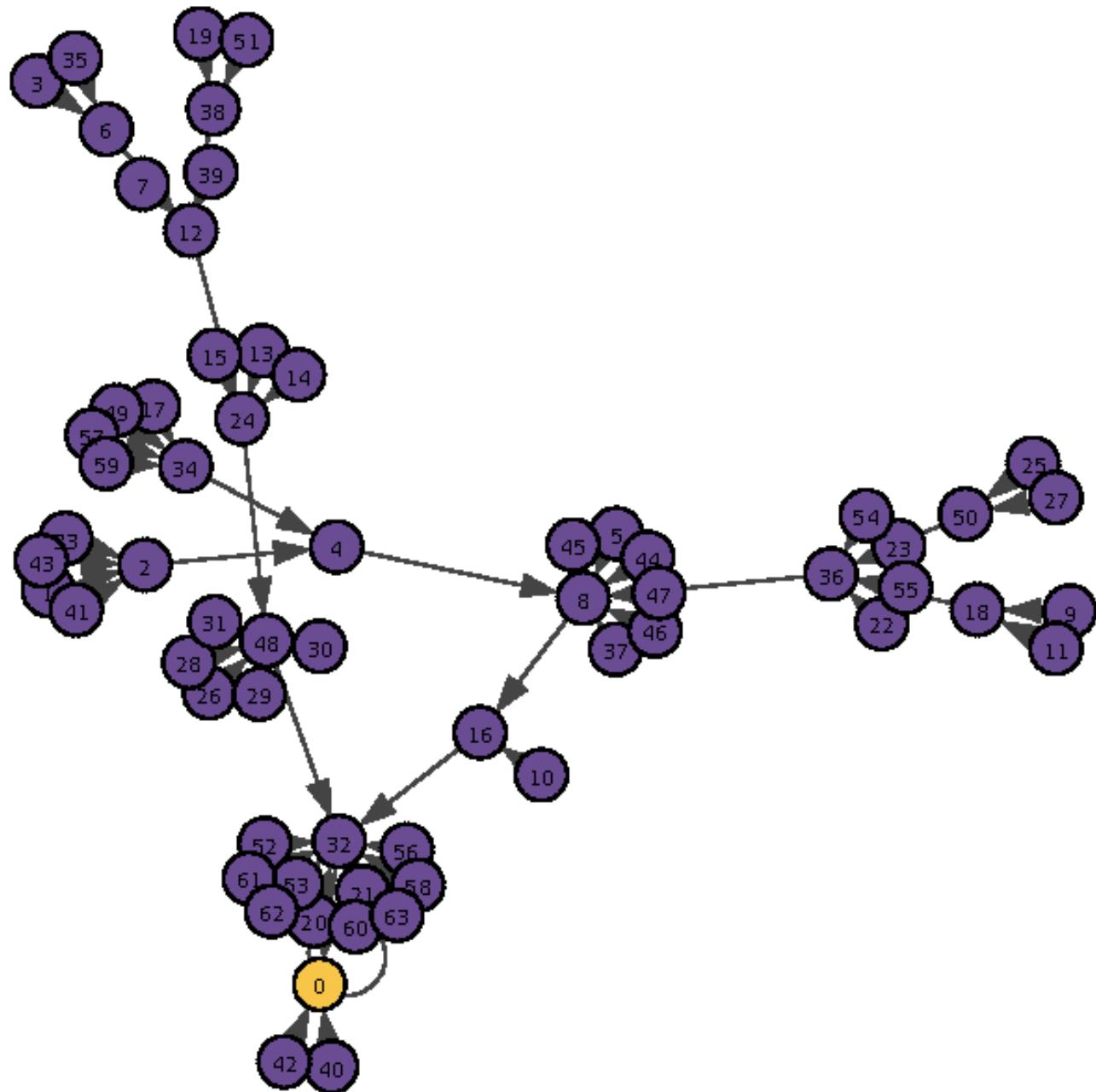


Figure 29: Grafo del atractor único 0 para la regla 10 con $n = 6$

3.1.12 Regla 11

Regla que genera árboles de atracción extensos compuestos por muchos nodos, lo que indica que el número de evoluciones requeridas para la estabilización del sistema es grande. Muestra un comportamiento similar a la regla 6.

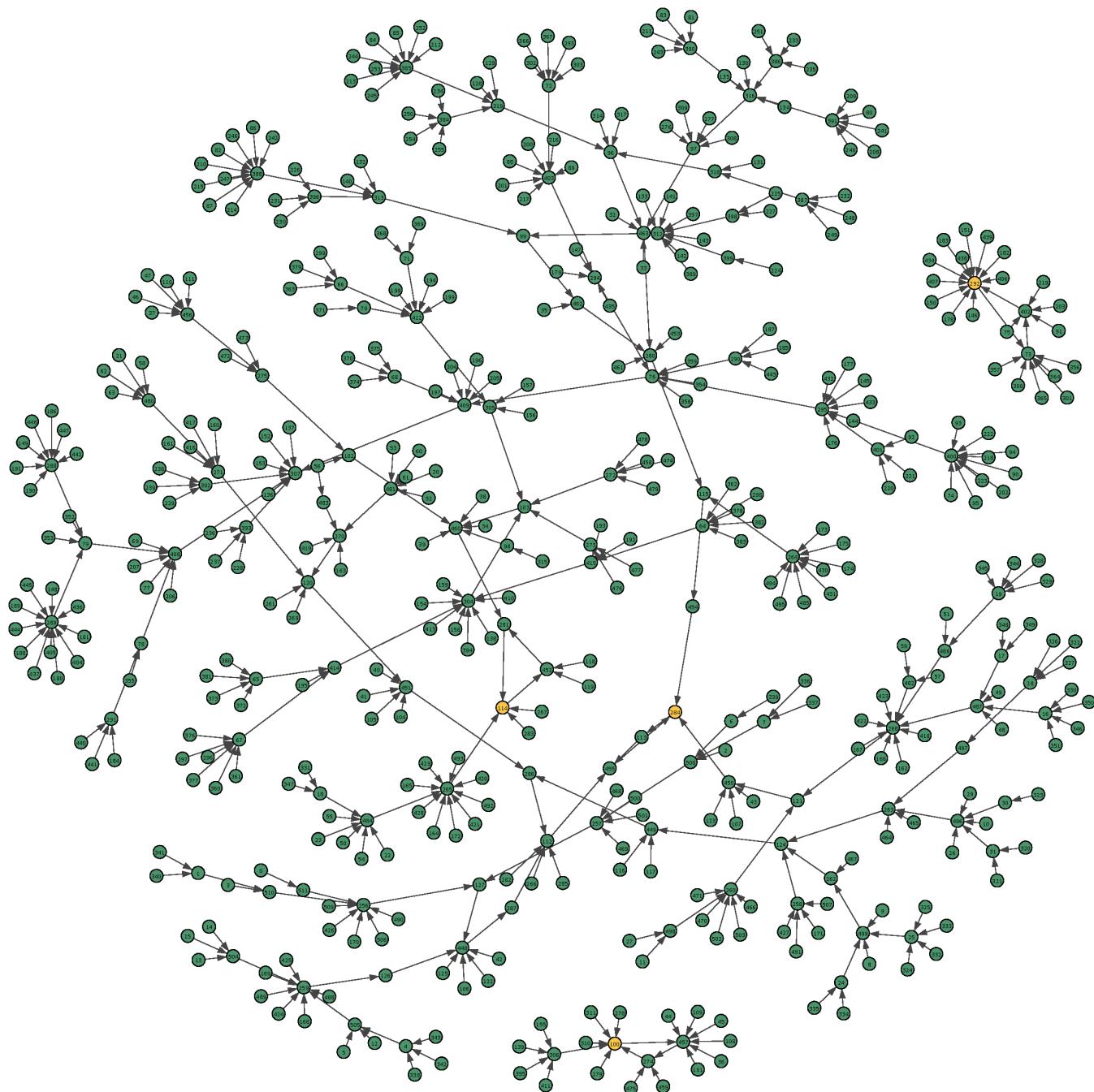


Figure 30: Grafo de los múltiples atractores para la regla 11 con $n = 9$

3.1.13 Regla 12

Regla de convergencia muy rápida, con todos los estados convergiendo a uno de los múltiples atractores en una sola evolución. También se destaca la presencia de algunos atractores marginales y comportamiento de los campos atractores similar a la regla 4.

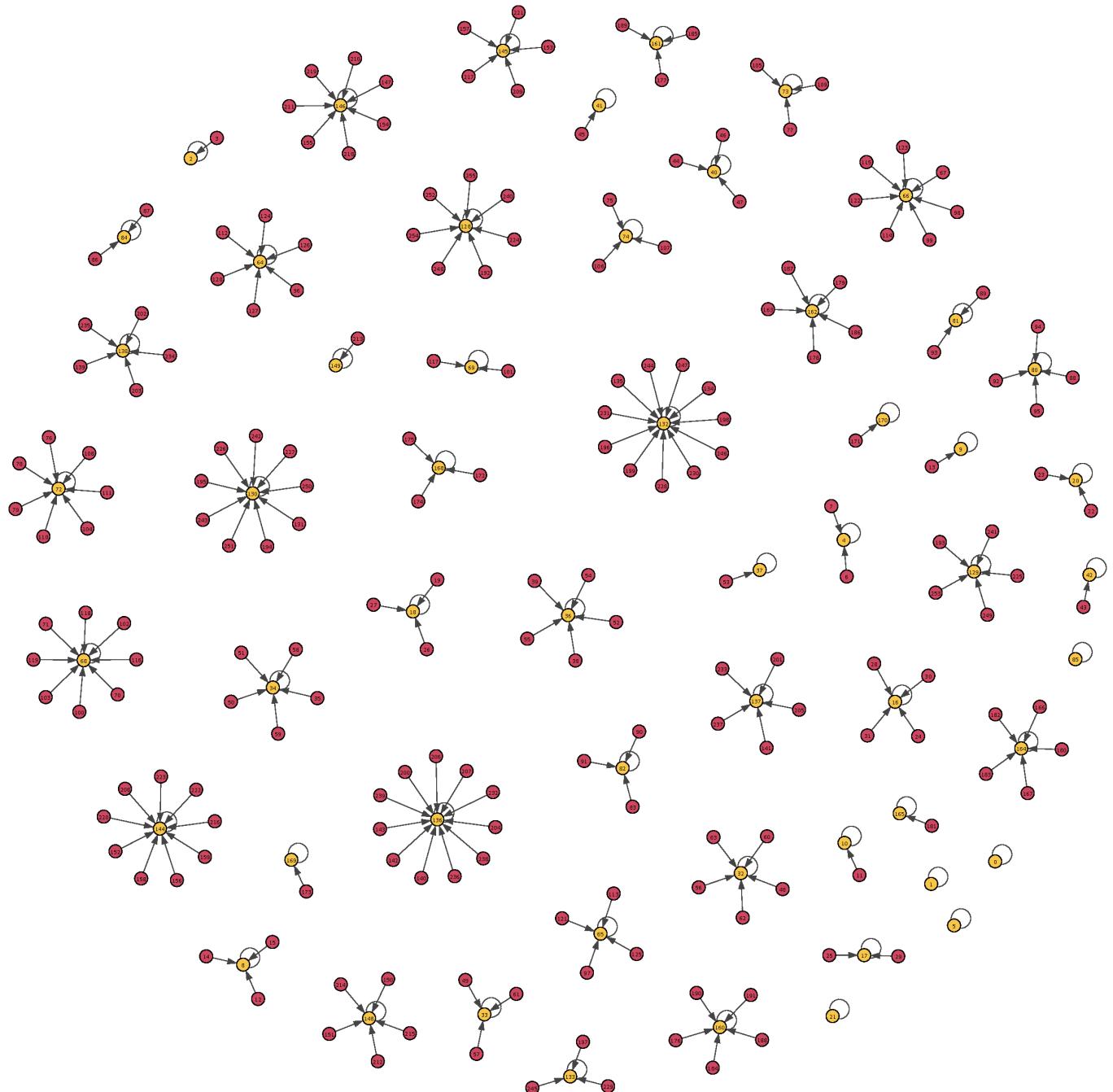


Figure 31: Grafo de los múltiples atractores para la regla 12 con $n = 8$

3.1.14 Regla 13

Regla que describe la creación de numerosos árboles de atracción con cantidad de atractores y una velocidad de convergencia baja.

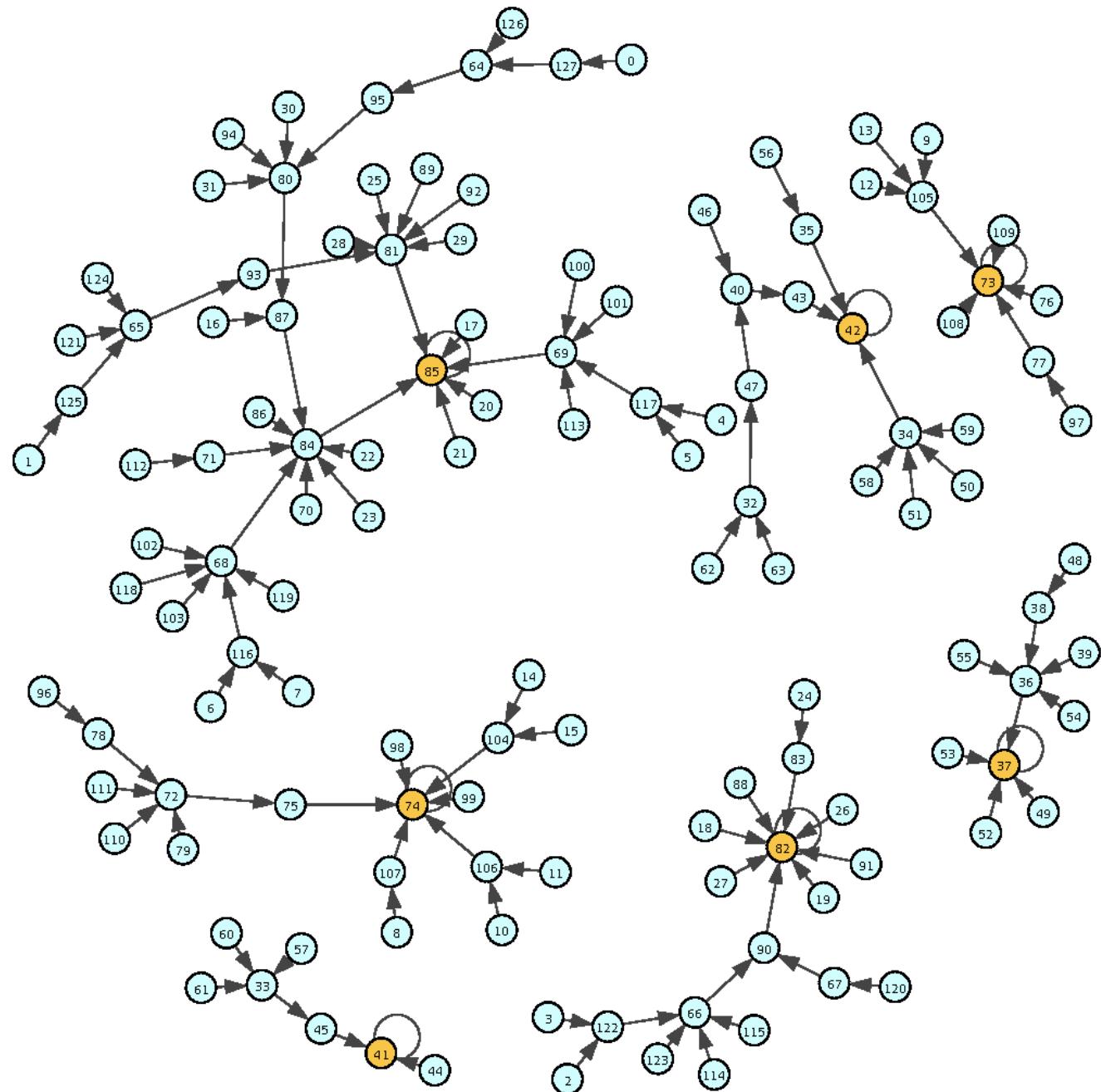


Figure 32: Grafo de los múltiples atractores para la regla 13 con $n = 7$

3.1.15 Regla 14

En la regla 14 se observa una producción de una cantidad moderada de árboles de atracción conformados por una gran cantidad de nodo, disminuyendo significativamente el número de evoluciones para alcanzar la estabilidad. Así también se observa la presencia de atractores marginales en cada una de las generaciones para diferentes valores de n .

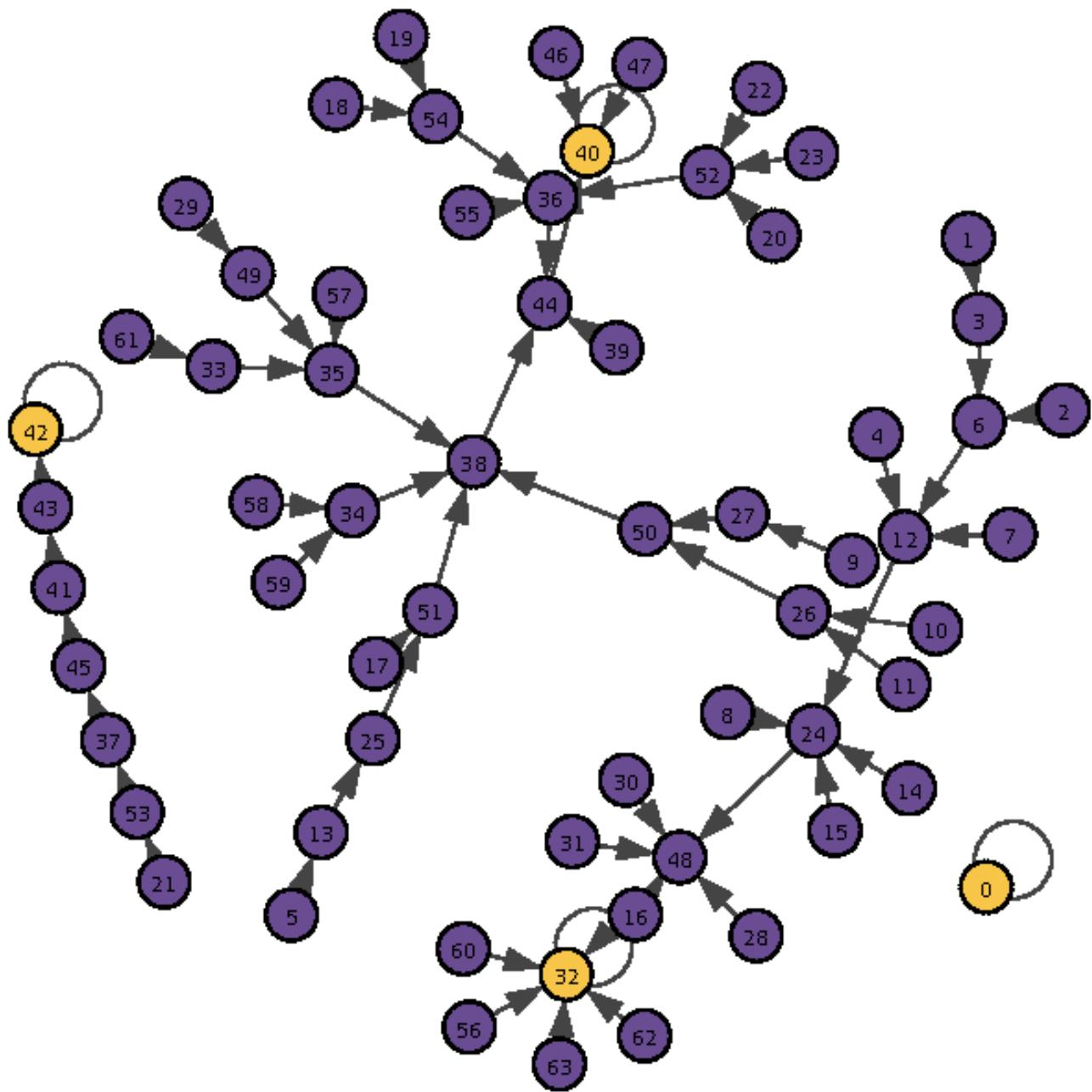


Figure 33: Grafo de los múltiples atractores para la regla 14 con $n = 6$

3.1.16 Regla 15

Con comportamiento similar a la regla 3 y 10, esta regla genera un árbol de atracción única con 1 solo atractor diferente de 0 para todos los valores n . Se observa una aglomeración en las ramas del árbol que indica una propiedad de atracción para los nodos extremos con una velocidad de convergencia moderada.

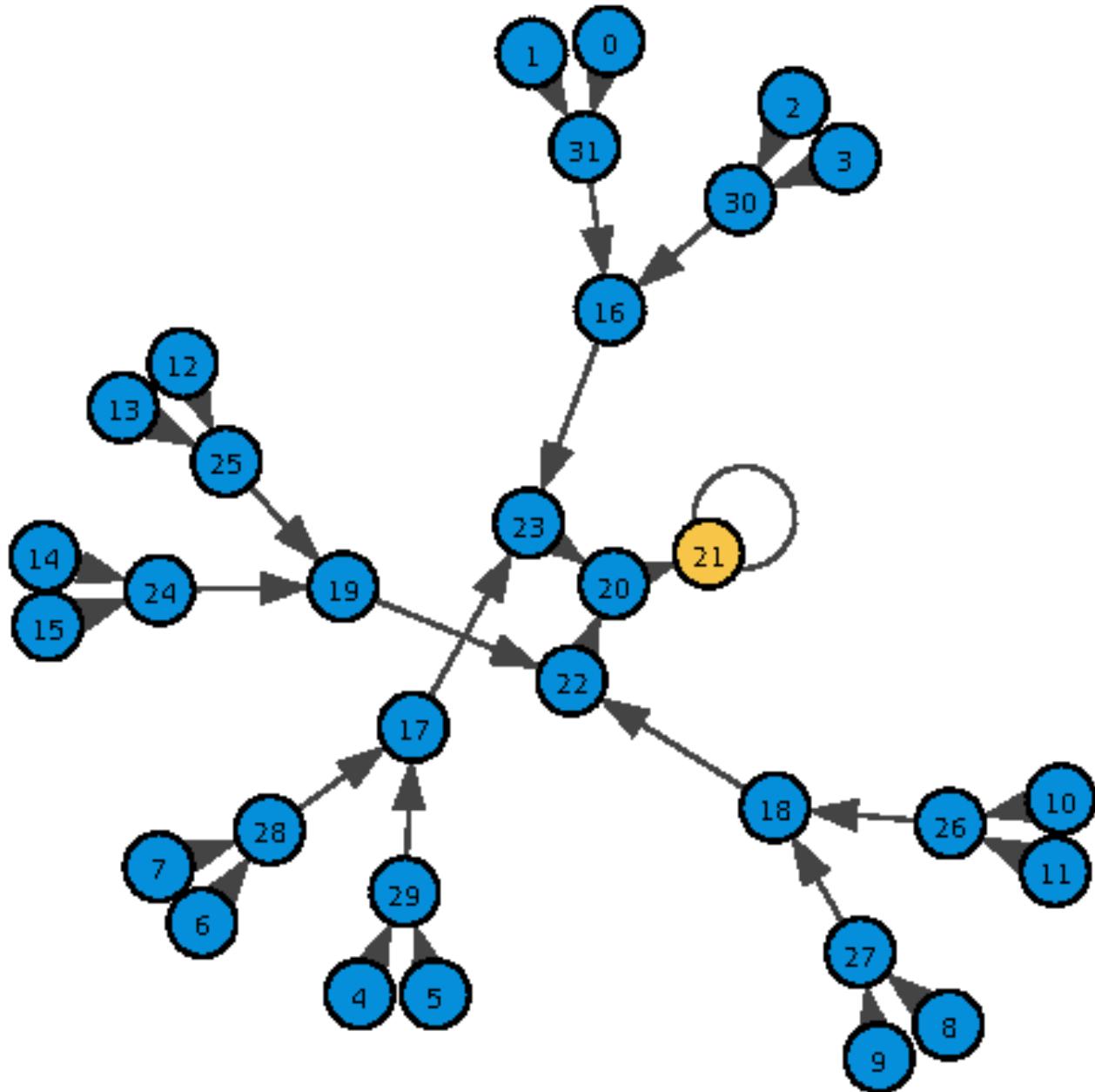


Figure 34: Grafo del atractor único para la regla 15 con $n = 5$

3.1.17 Regla 18

Producción de árboles de atracción extensos con nodos que tienen propiedades atractoras, con presencia menor de árboles pequeños de unos pocos nodos. Se rescata la presencia dominante del atractor 0 donde existe una importante concentración de estados que convergen en 1 sola evolución.

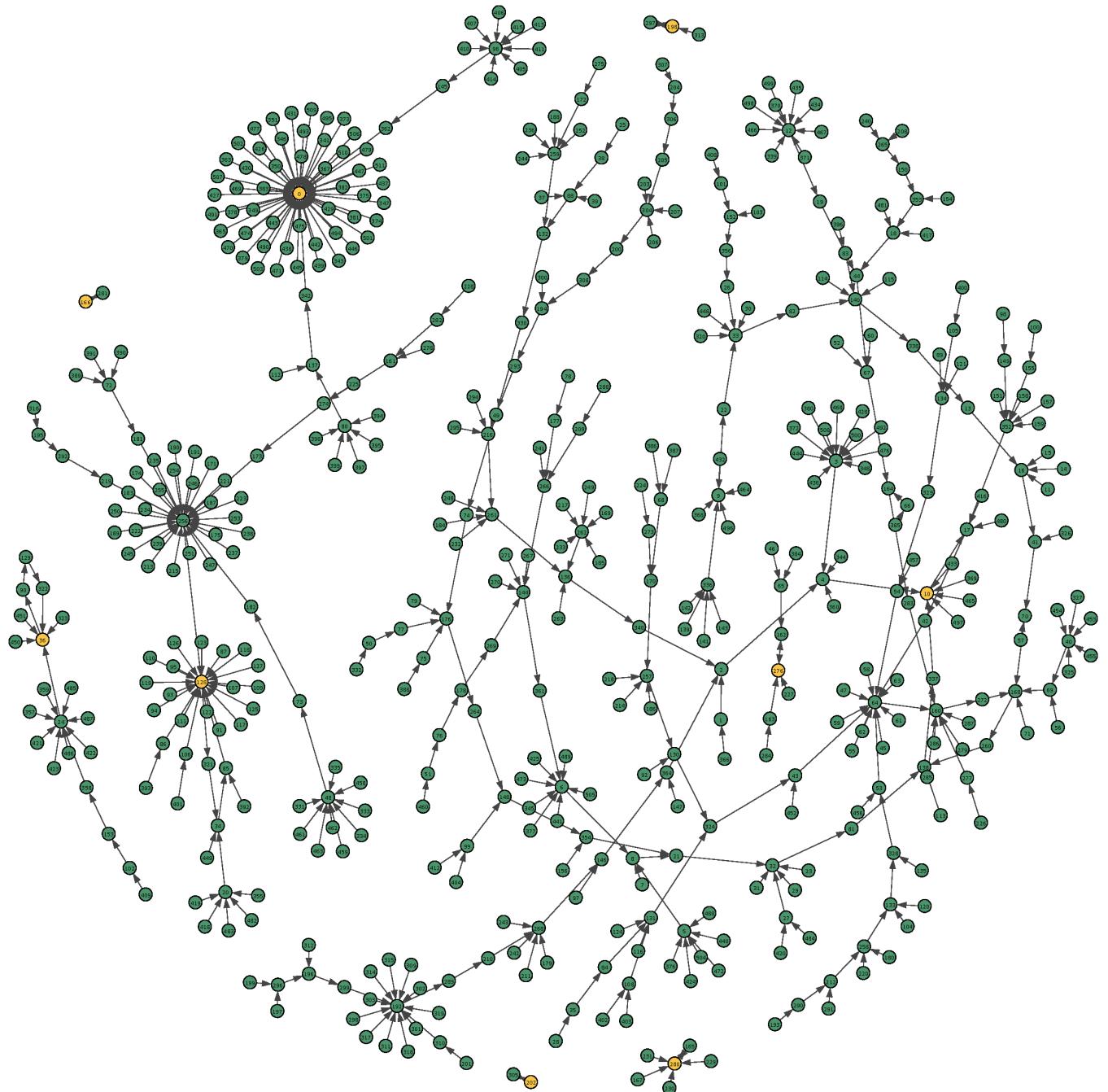


Figure 35: Grafo de los múltiples atractores para la regla 18 con $n = 9$

3.1.18 Regla 19

Comportamiento con árboles de atracción que convergen en unas pocas etapas de evolución al nodo atractor 0 con la presencia aislada de algunos árboles de evolución un poco más extensos. Destaca la presencia del nodo atractor 0 con una mayoría de nodos ancestros.

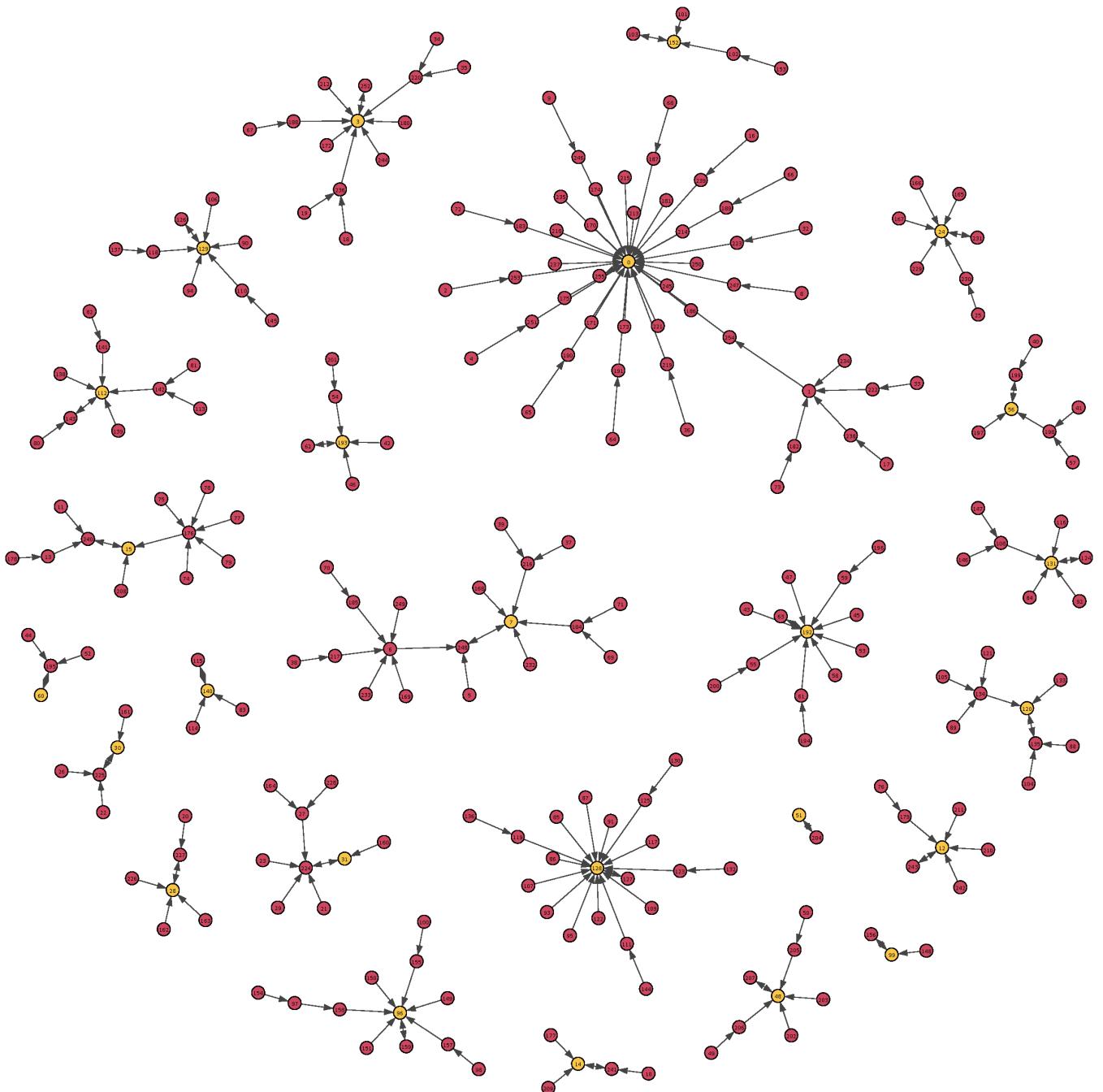


Figure 36: Grafo de los múltiples atractores para la regla 19 con $n = 8$

3.1.19 Regla 22

Regla que se describe con una mayor producción de árboles de atracción conformados extensamente con un gran número de nodos y por lo tanto una velocidad de convergencia baja. Aparece en valores para n impares, 1 único nodo atractor marginal con estado variante.

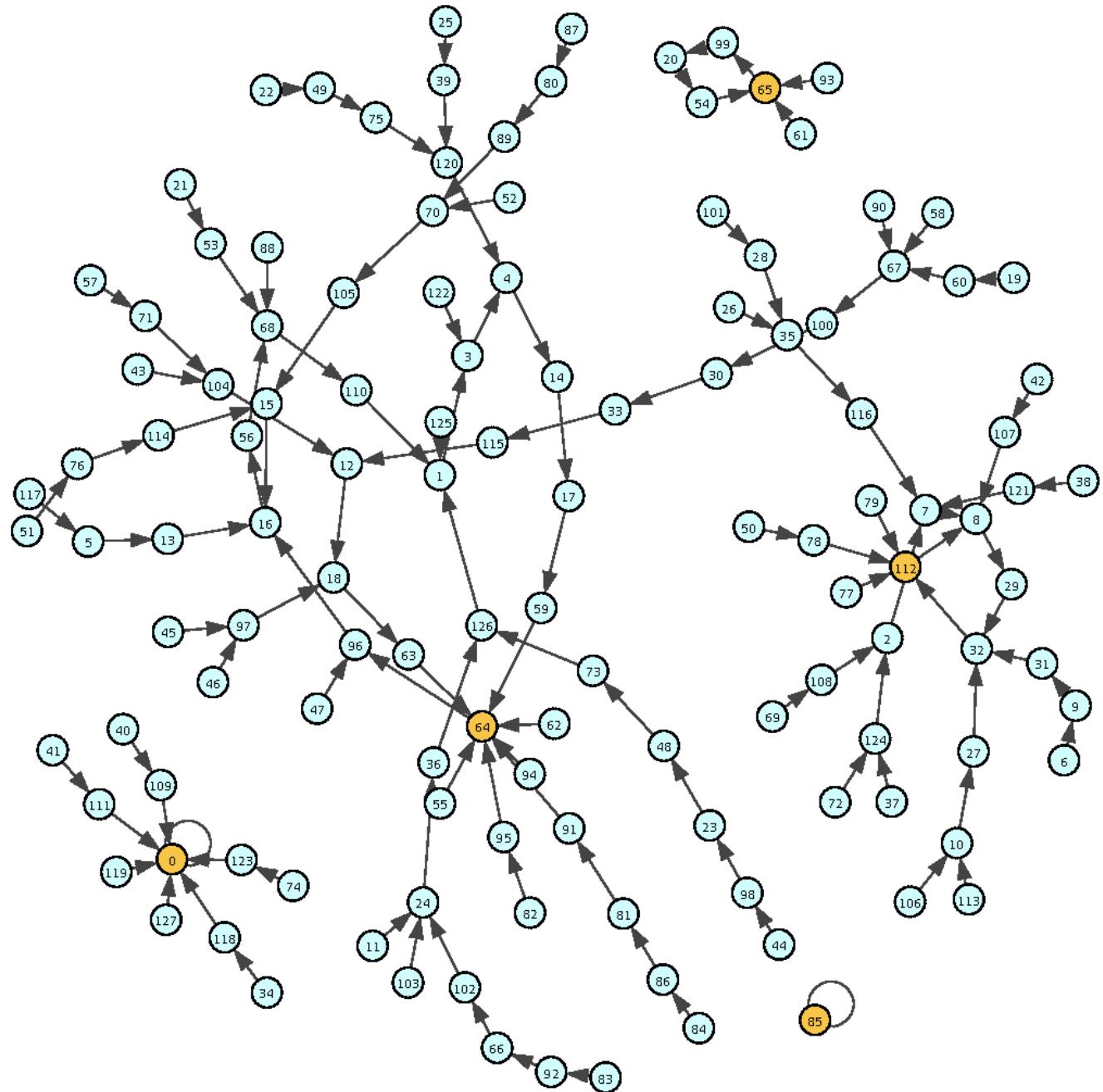


Figure 37: Grafo de los múltiples atractores para la regla 22 con $n = 7$

3.1.20 Regla 23

Comportamiento similar a la regla 22, esta regla genera árboles de atracción, con diferencia en que el número de estos es mucho mayor en comparación. Constantemente se observa la presencia de un atractor marginal, y árboles de pocas configuraciones.

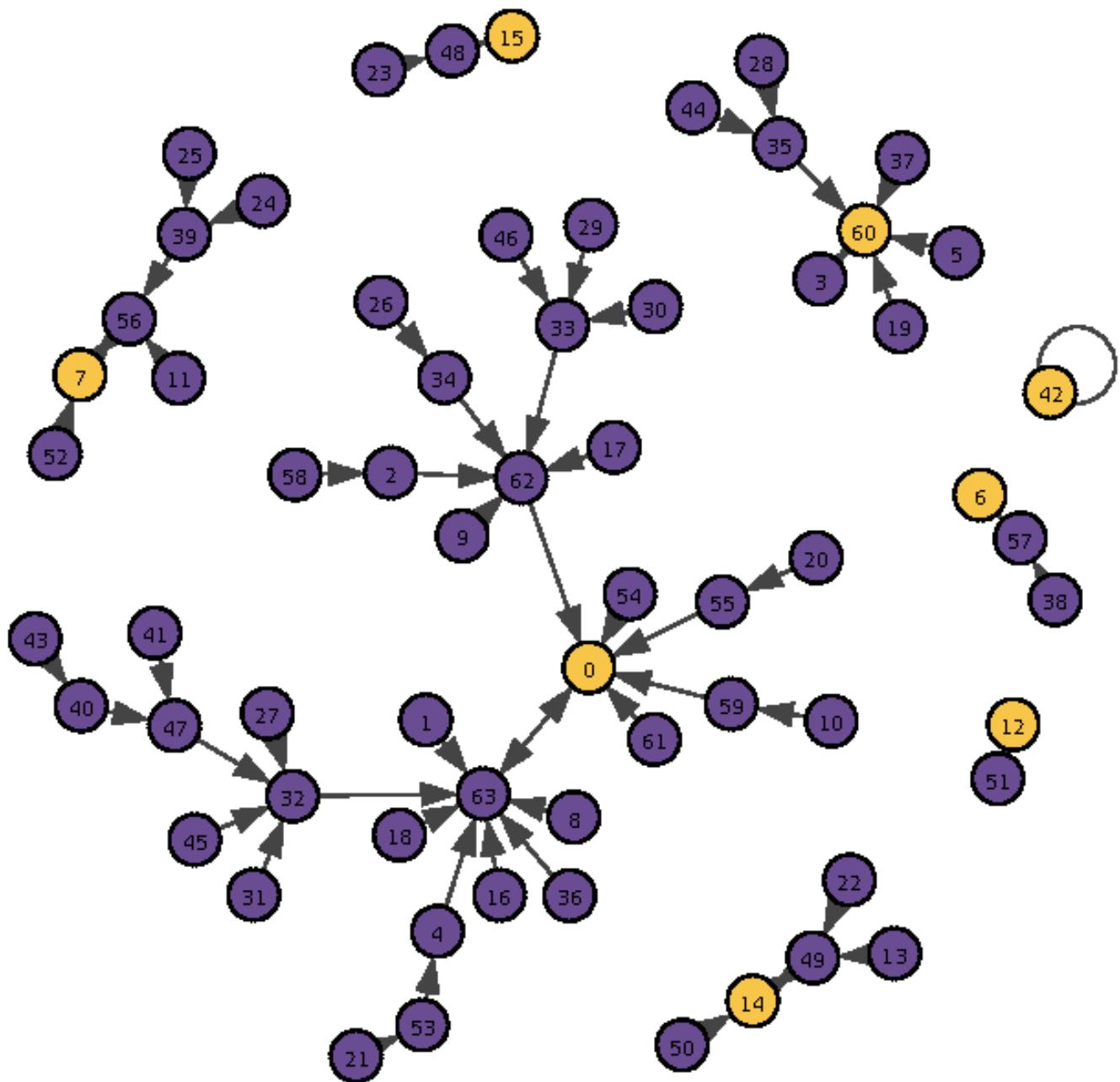


Figure 38: Grafo de los múltiples atractores para la regla 23 con $n = 6$

3.1.21 Regla 24

Árbol de atracción único con convergencia al estado 0 y nodos en los extremos de las ramas con propiedades de atracción. Similar a las reglas 10 y 3.

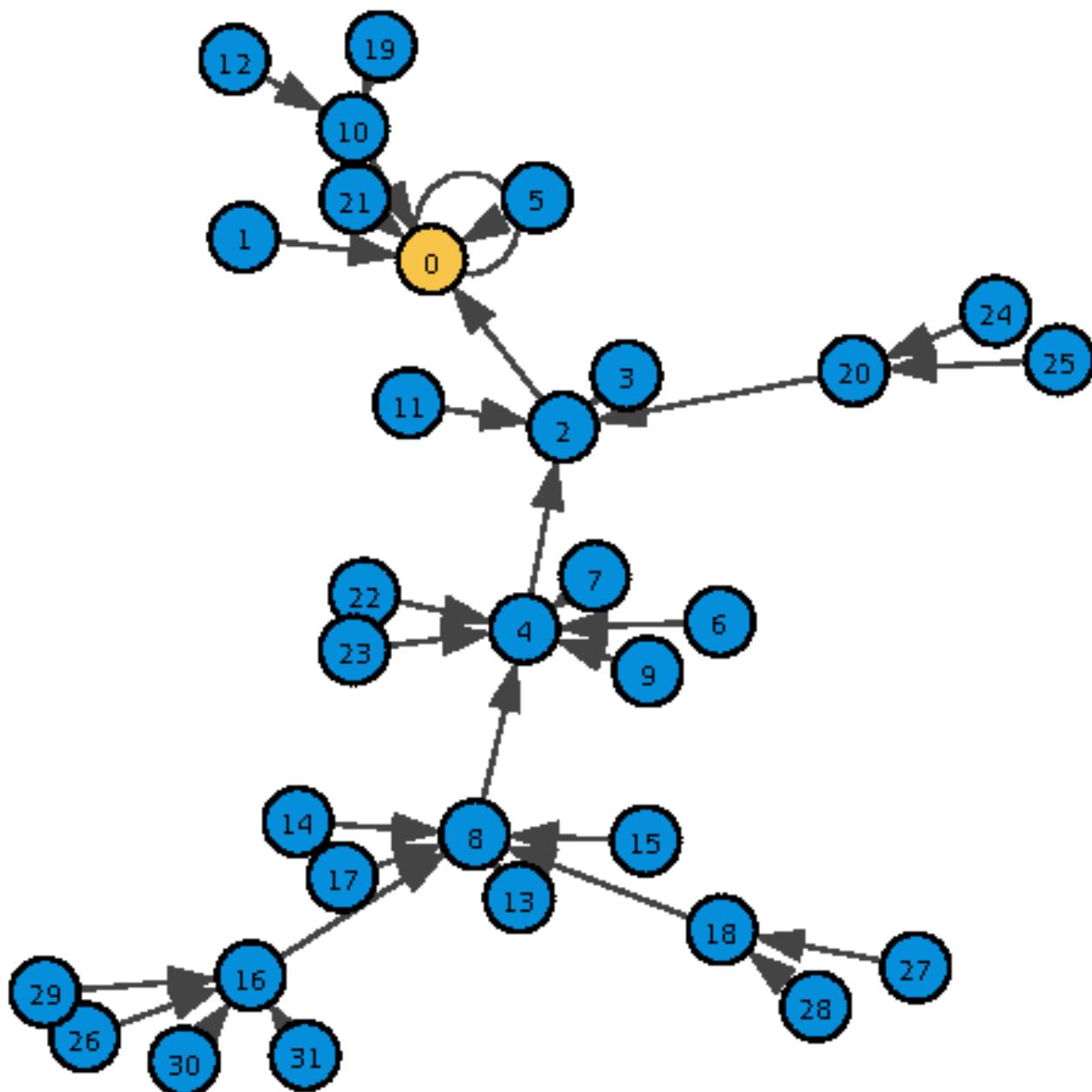


Figure 39: Grafo del atractor único para la regla 24 con $n = 5$

3.1.22 Regla 25

La regla 25 produce árboles de atracción en cantidad bastante limitada y con una extensión importante, indicando directamente un gran número de fases de evolución para alcanzar la estabilidad en el estado atractor.

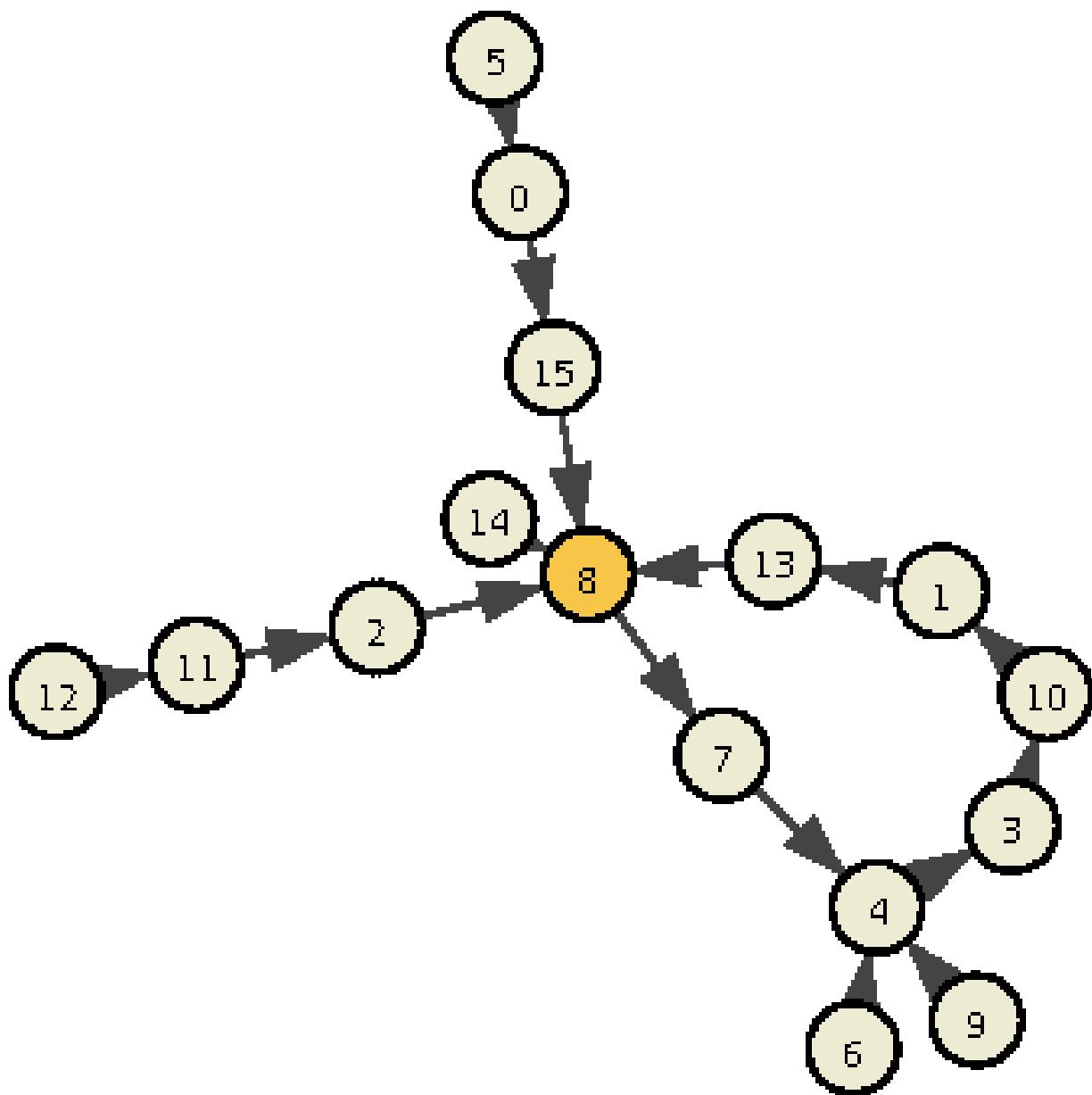


Figure 40: Grafo del atractor único para la regla 25 con $n = 4$

3.1.23 Regla 26

Con un comportamiento poco convencional en la descripción de los campos de atracción, la regla 26 genera árboles de evolución en diferentes cantidades para los valores de n . En algunas genera hasta 3 árboles de atracción, en otros como para $n=8,9$ se generan únicamente 2 árboles y se alcanza un máximo de 5 atractores para $n=12$.

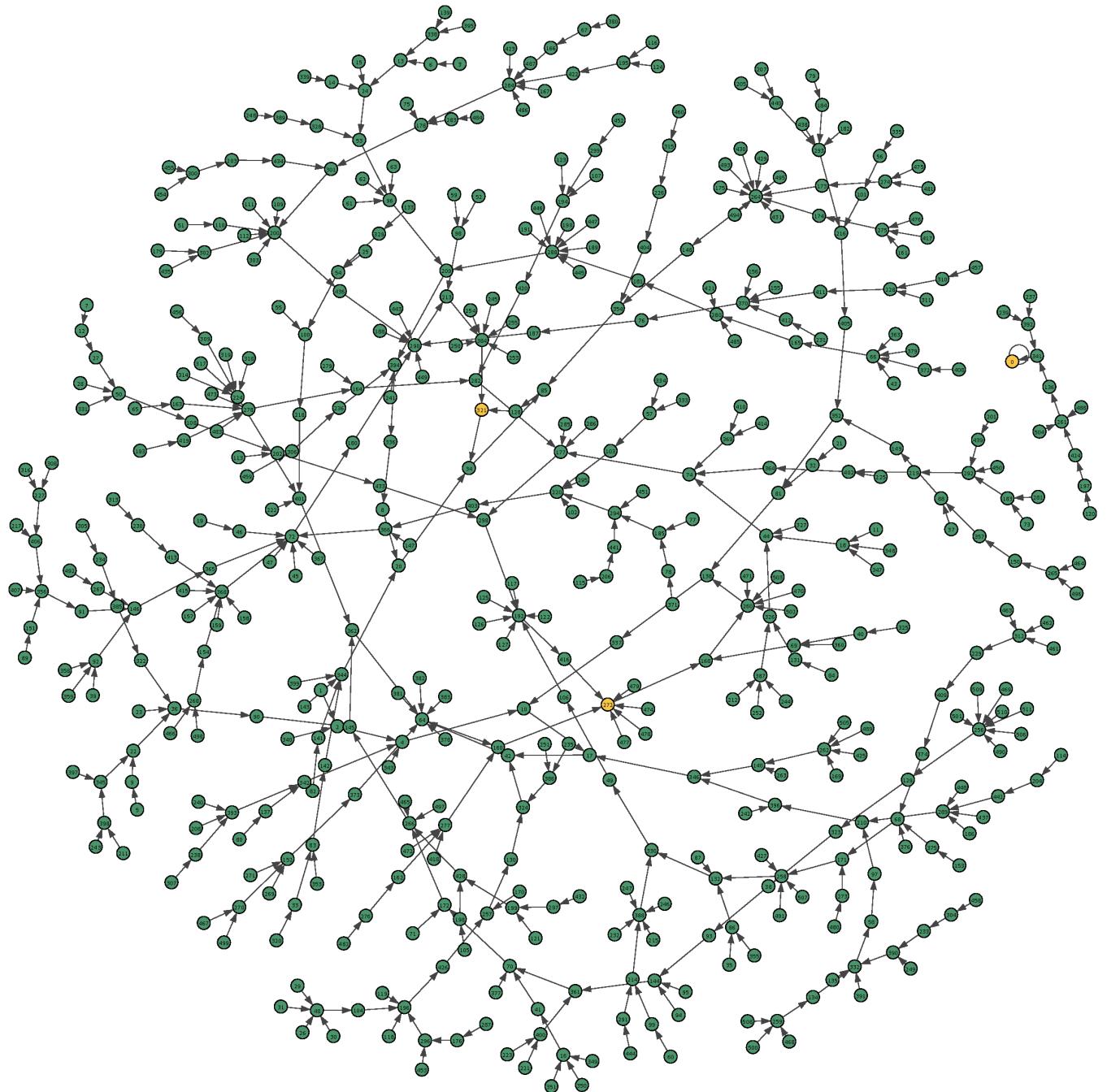


Figure 41: Grafo de los atractores para la regla 26 con $n = 9$

3.1.24 Regla 27

Describo con árboles de atracción en cantidad pequeña (máximo 3 para $n=15$), se identifica 1 de estos árboles de evolución con una gran extensión y conformación de nodos, mientras que el otro se mantiene muy pequeño a tan solo algunos nodos de extensión. Importante mencionar que a medida que aumenta el valor de n así también lo hace el número de árboles, describiendo un crecimiento muy lento.

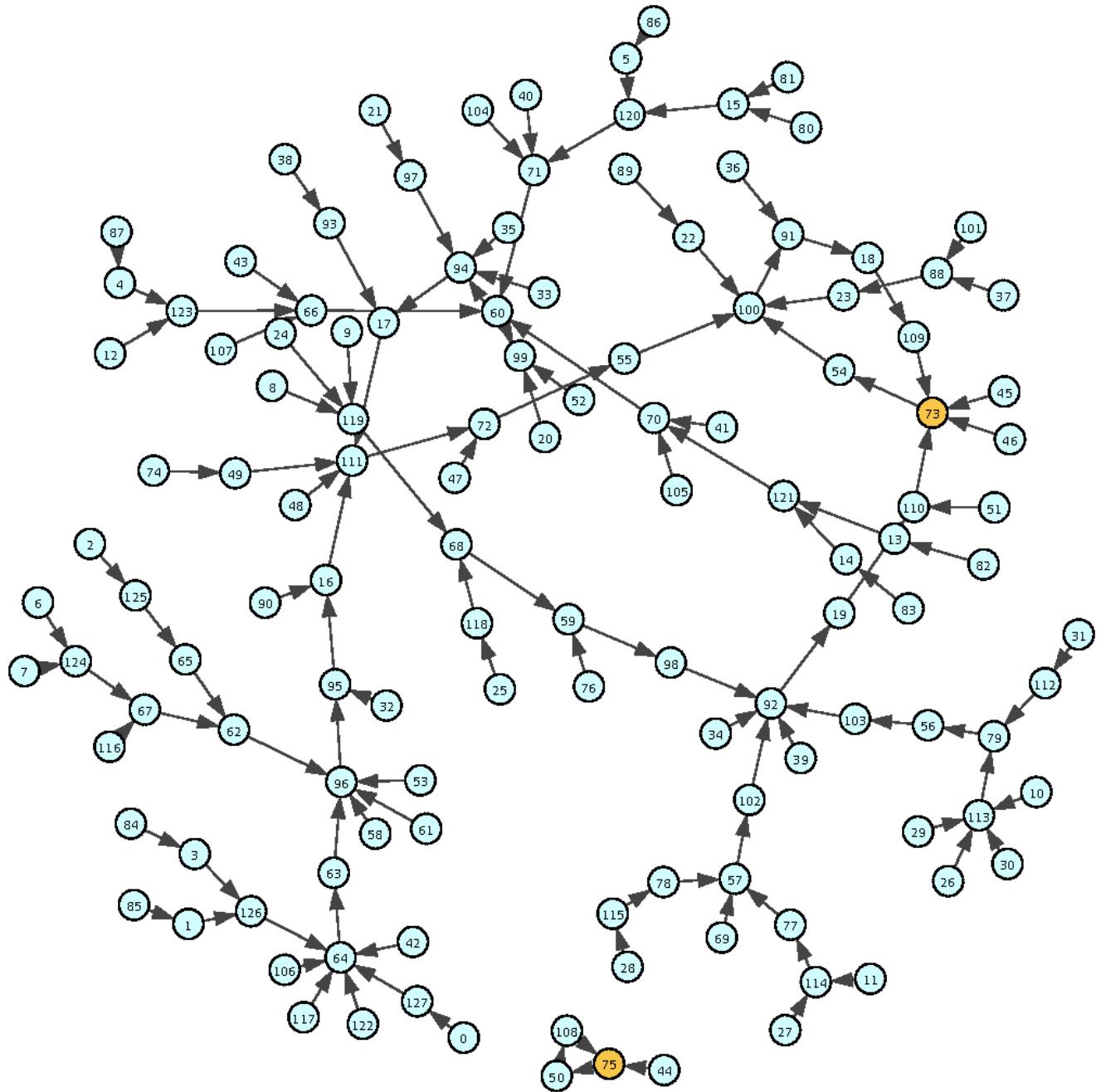


Figure 42: Grafo de los atractores para la regla 27 con $n = 7$

3.1.25 Regla 28

Producción de árboles de atracción con un aumento en su número a medida que aumenta el valor de n acelerada, aunque a causa de esto, la extensión de los árboles no es muy grande pero bien distribuida. Así también se reporta la observación constante de un par de atractores marginales que siempre pertenecen a los estados 0 y 1.

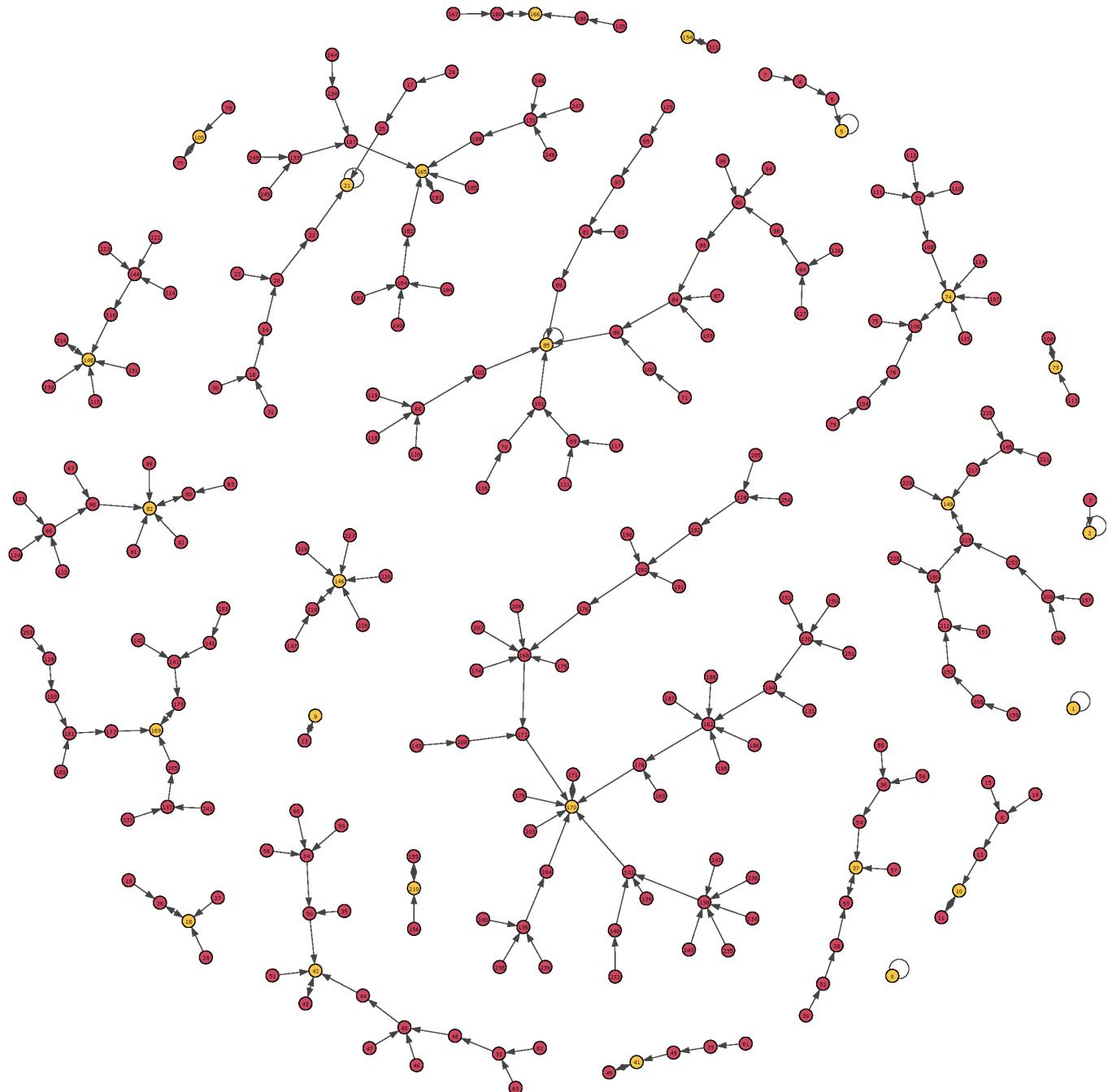


Figure 43: Grafo de los atractores para la regla 28 con $n = 8$

3.1.26 Regla 29

Alta producción de árboles conformados por unos pocos nodos, resultando en un conjunto grande de atractores a los que los demás estados convergen, pero asegura la velocidad de estabilización. La conformación de pequeños árboles de evolución es tan alta que alcanza un máximo de 3,526 cuando $n=15$.

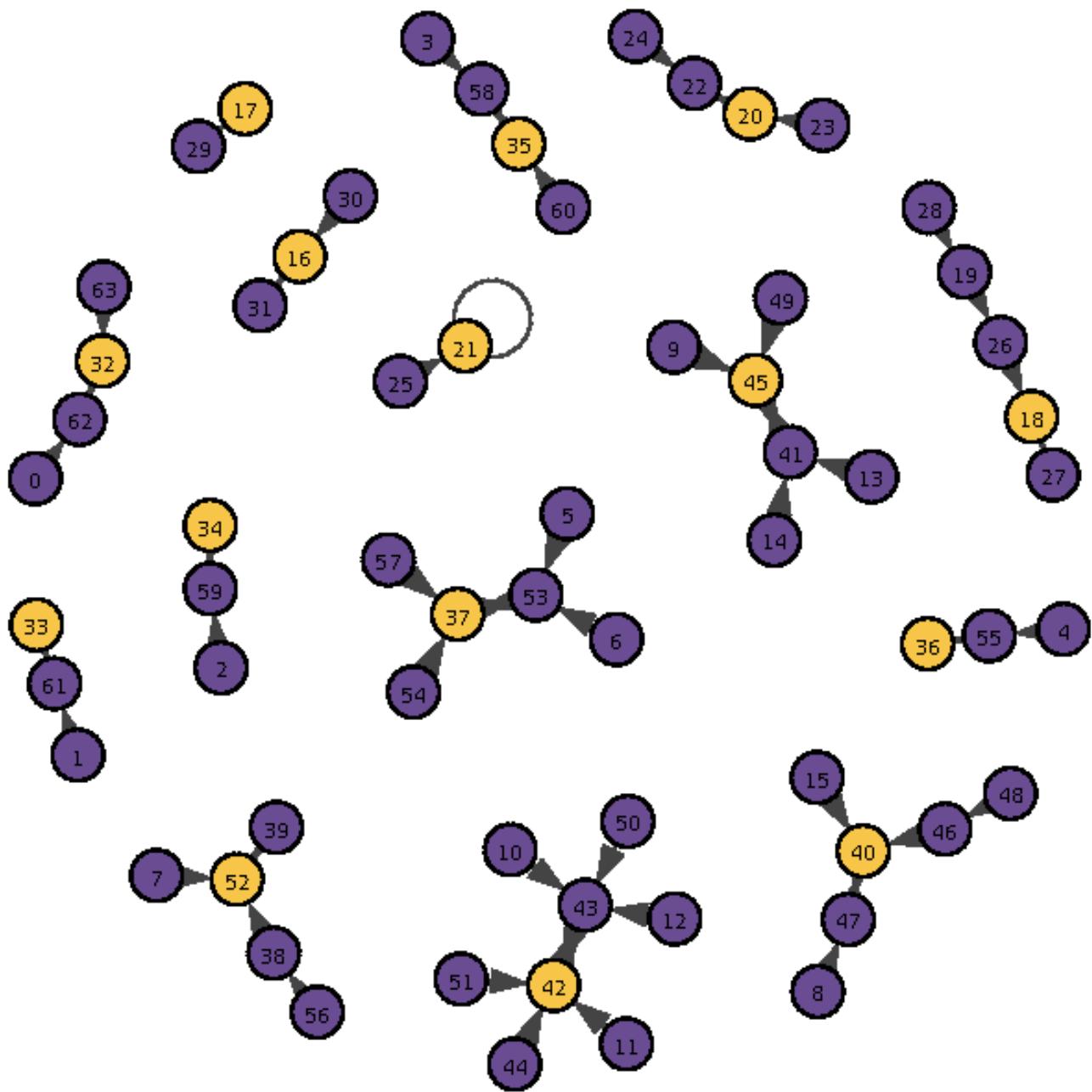


Figure 44: Grafo de los atractores para la regla 29 con $n = 6$

3.1.27 Regla 30

Con comportamiento similar a la regla 27, intercala entre la producción de 1 a 2 árboles de evolución y el estado 0 como atractor marginal. Uno de los árboles es siempre significativamente más grande que el otro, más no su distribución es bastante uniforme por lo que las hojas del árbol no tienen propiedades de atracción significativas.

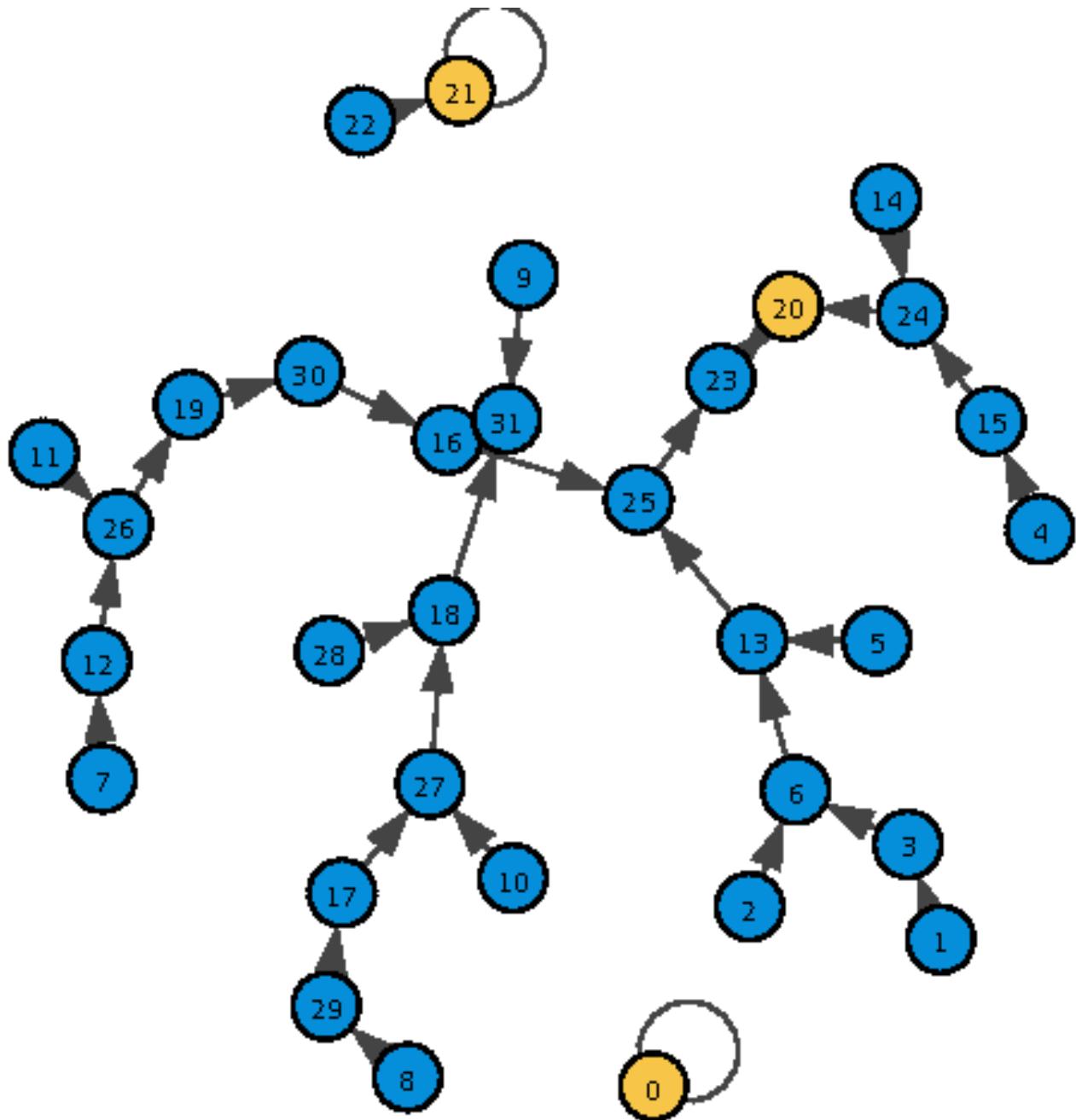


Figure 45: Grafo de los atractores para la regla 30 con $n = 5$

3.1.28 Regla 32

Regla sumamente interesante que genera un árbol de evolución único con las hojas en los extremos de las ramas con propiedades de atracción, pero siempre manteniendo la convergencia para el nodo 0 el cual se encuentra constante en el centro de todas las ramas evolutivas. Finalmente se rescata una relativamente alta velocidad de convergencia pues las ramas no se encuentran conformadas por un gran número de nodos consecutivos.

Esta regla genera gráficamente debido a la distribución de sus ramas, figuras de grafos muy particulares, que en algunos de los casos resultan topológicamente simétricas si se realiza un corte central

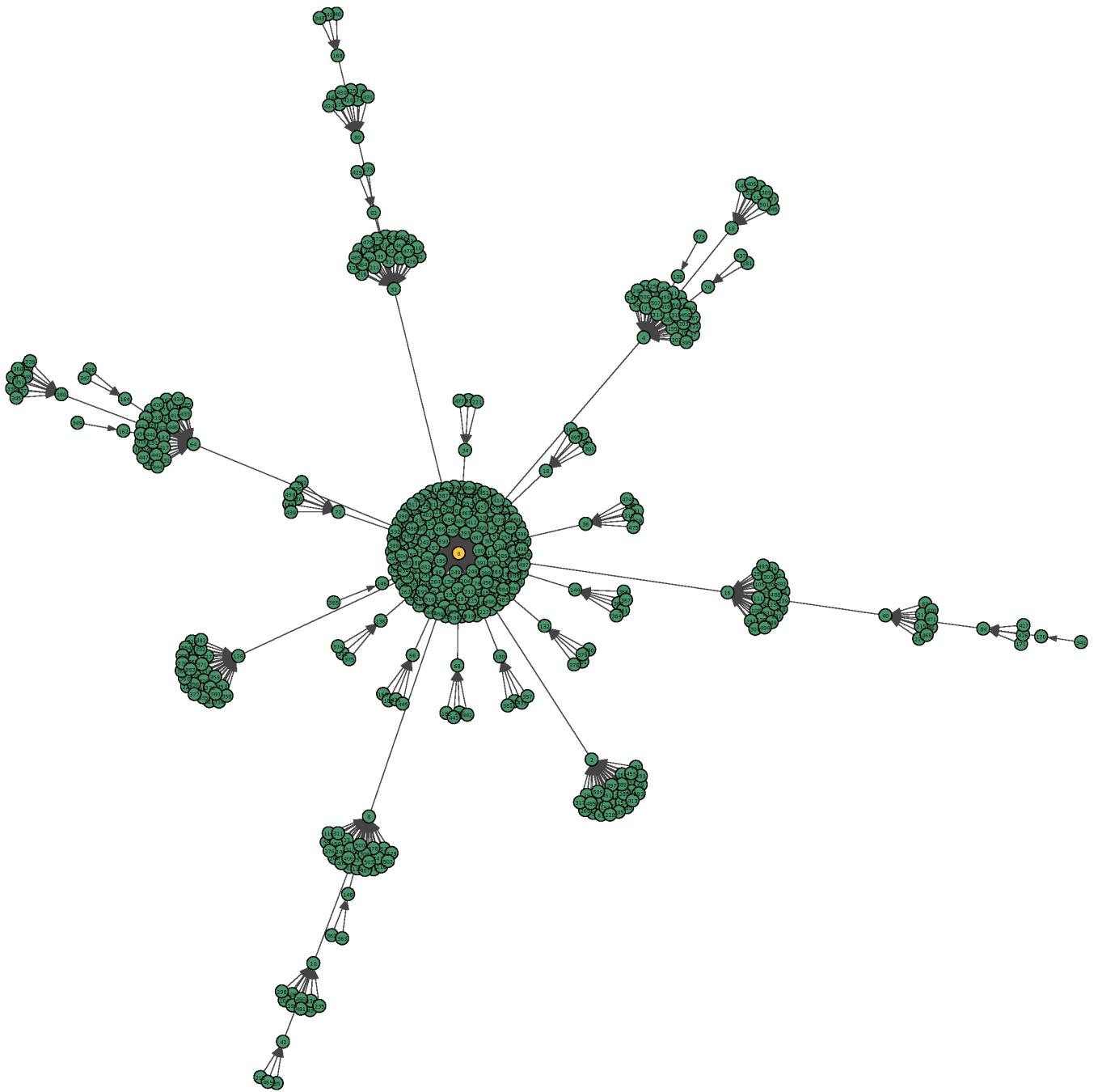


Figure 46: Grafo de los atractores para la regla 32 con $n = 9$

3.1.29 Regla 33

Producción de árboles de evolución compactos y en número considerable, con una gran predominancia por parte de los nodos atractores, por lo que el número de evoluciones requeridas para la convergencia no es muy grande.

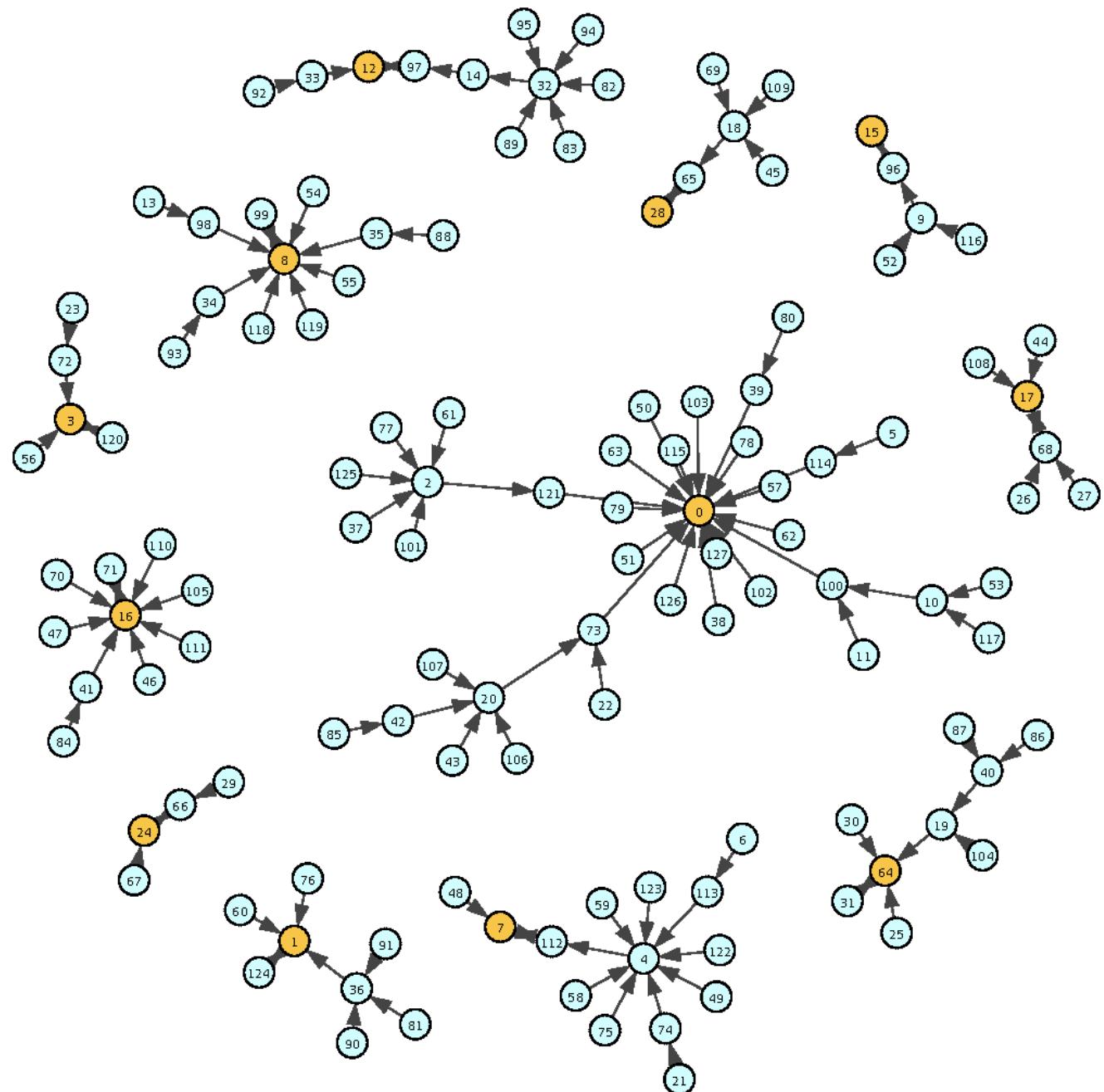


Figure 47: Grafo de los atractores para la regla 33 con $n = 7$

3.1.30 Regla 34

Otra regla con producción de árbol de evolución único con hojas en extremos de las ramas con propiedades atractoras pero con convergencia a un único atractor, es estado 0.

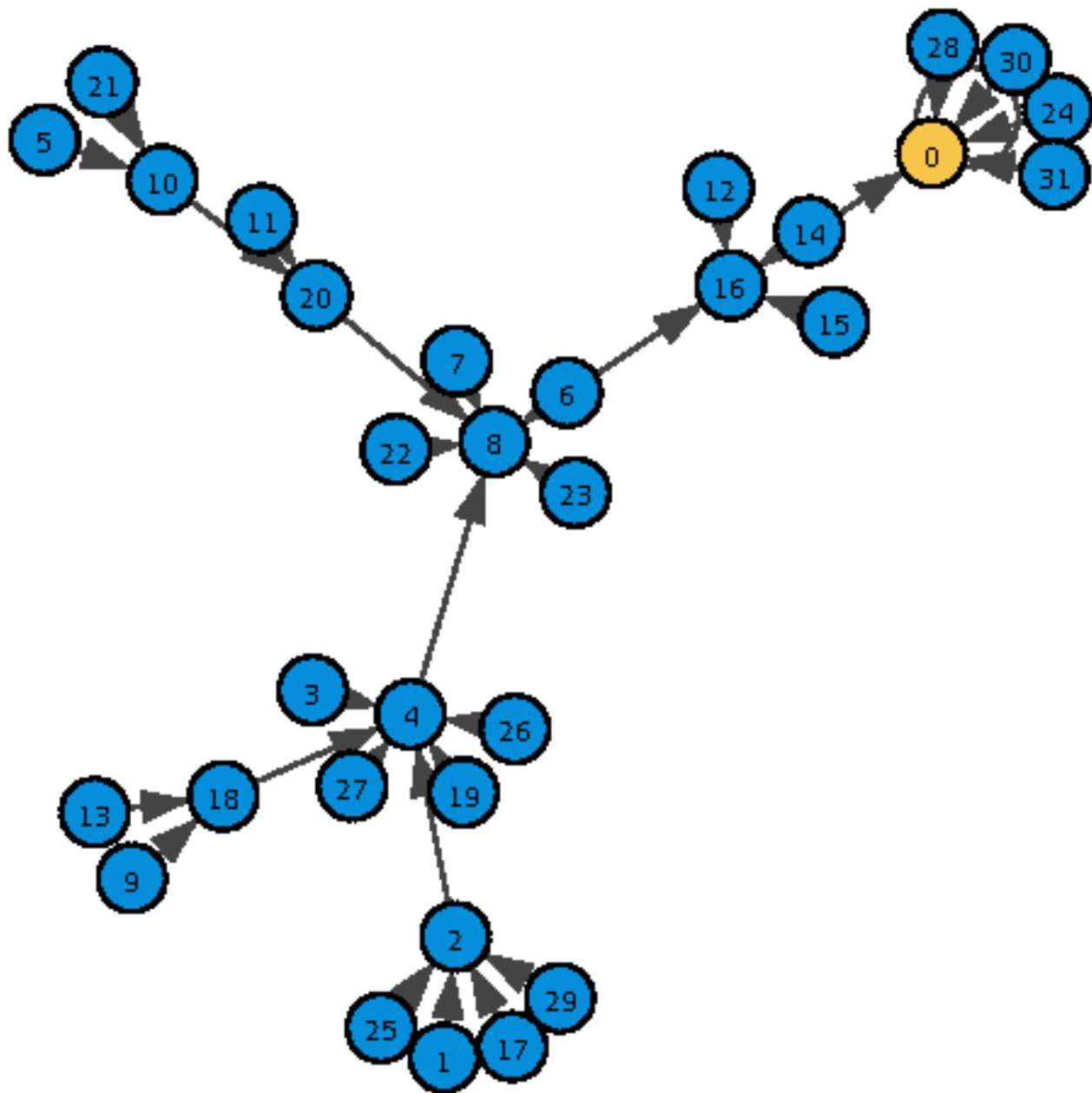


Figure 48: Grafo de los atractores para la regla 34 con $n = 5$

3.1.31 Regla 35

Producción con tendencia lenta pero creciente de árboles de atracción con una distribución que permite la extensión de los árboles y previene las propiedades de atracción a otros nodos que no sean el atractor, esto influye directamente en el número de evoluciones requeridas para la estabilización del sistema, que será un número medianamente alto.

Se observa una particularidad no observada hasta el momento en reglas anteriores, donde existe una tendencia generalizada donde el nodo atractor forme parte en un ciclo único compuesto por exclusivamente 2 estados. Si bien todos los atractores forman parte de un ciclo dentro del árbol de atracción, el conjunto de nodos que conforman el ciclo varía en extensión, y sin embargo en esta regla es constante la cardinalidad par.

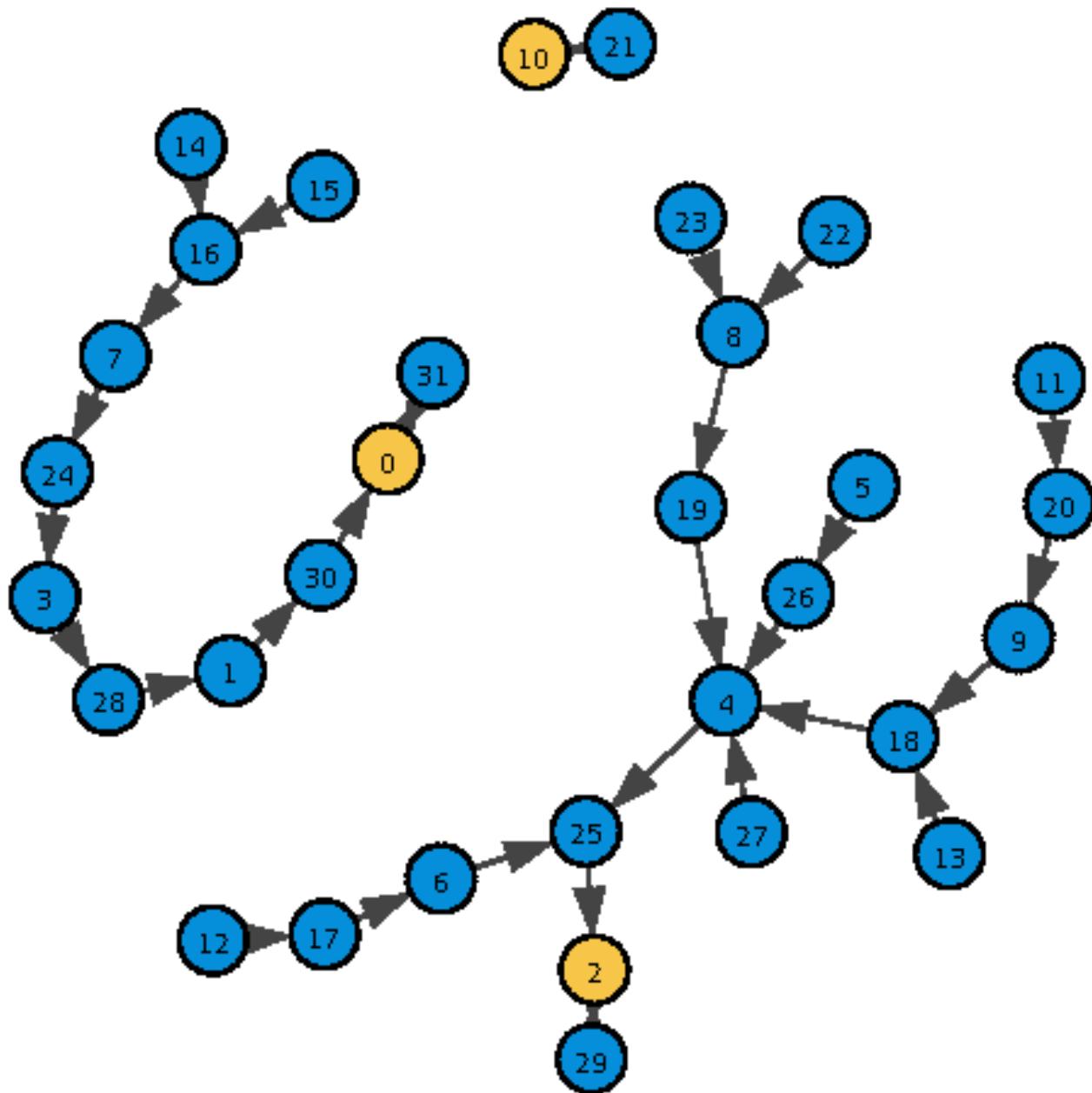


Figure 49: Grafo de los atractores para la regla 35 con $n = 5$

3.1.32 Regla 36

Construcción limitada de árboles de evolución con pocas etapas de evolución hasta la convergencia del árbol. Se muestra predominante el atractor con el estado 0 para todos los casos de valores de n , y se rescata finalmente la presencia de algunos atractores marginales que aumentan sus números lentamente a medida que n también crece.

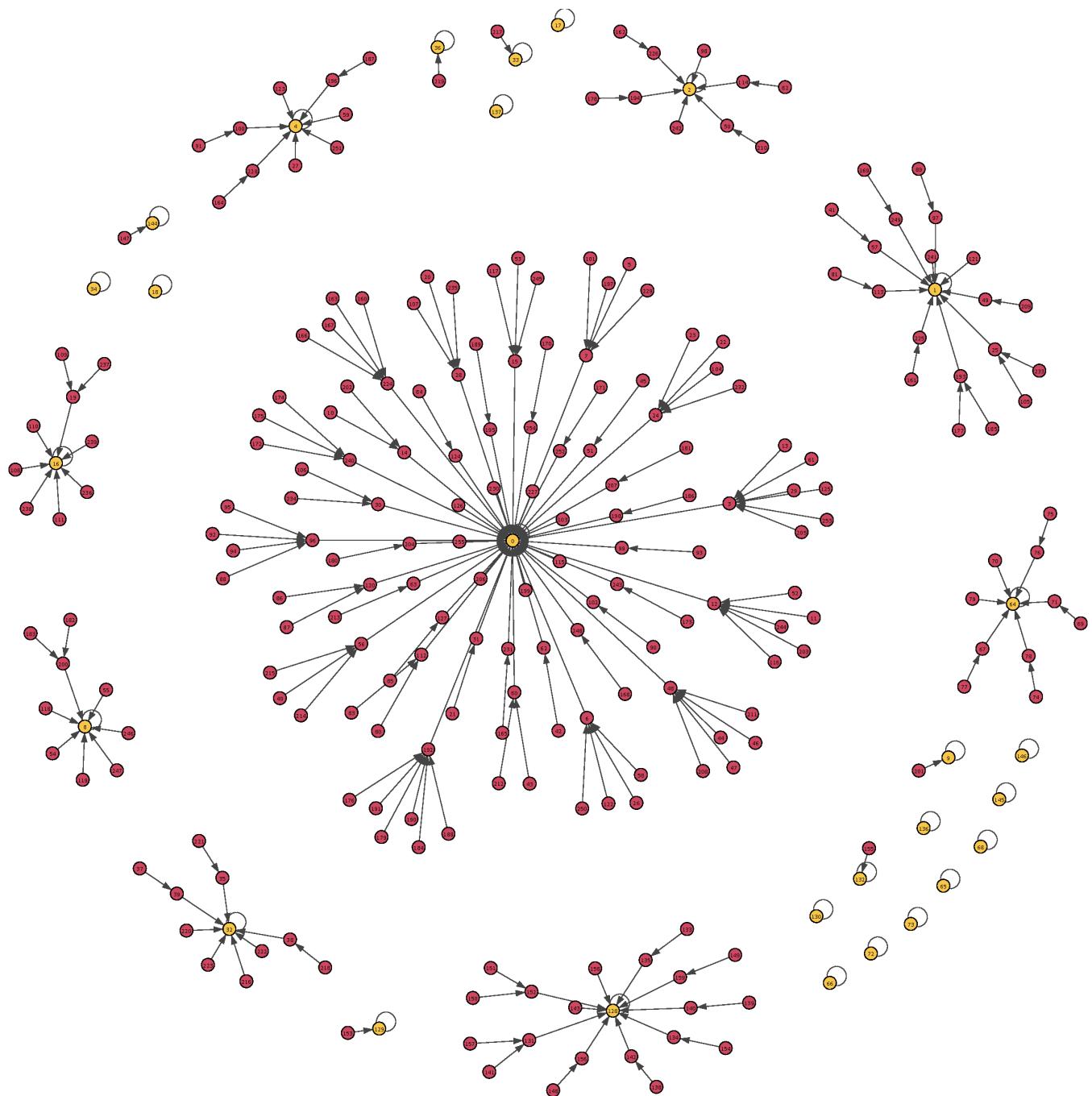


Figure 50: Grafo de los atractores para la regla 36 con $n = 8$

3.1.33 Regla 37

Regla que describe sus campos de atracción con la conformación de árboles de evolución en una tendencia creciente, según aumenta el valor de n , con una distribución que permite su extensión aunque no elimina del todo las propiedades atractoras de algunos nodos extremos de las ramas, aunque esta es moderada y no se nota una concentración fuerte de nodos en estos extremos. También se destaca la presencia errática de atractores marginales pero en números muy limitados

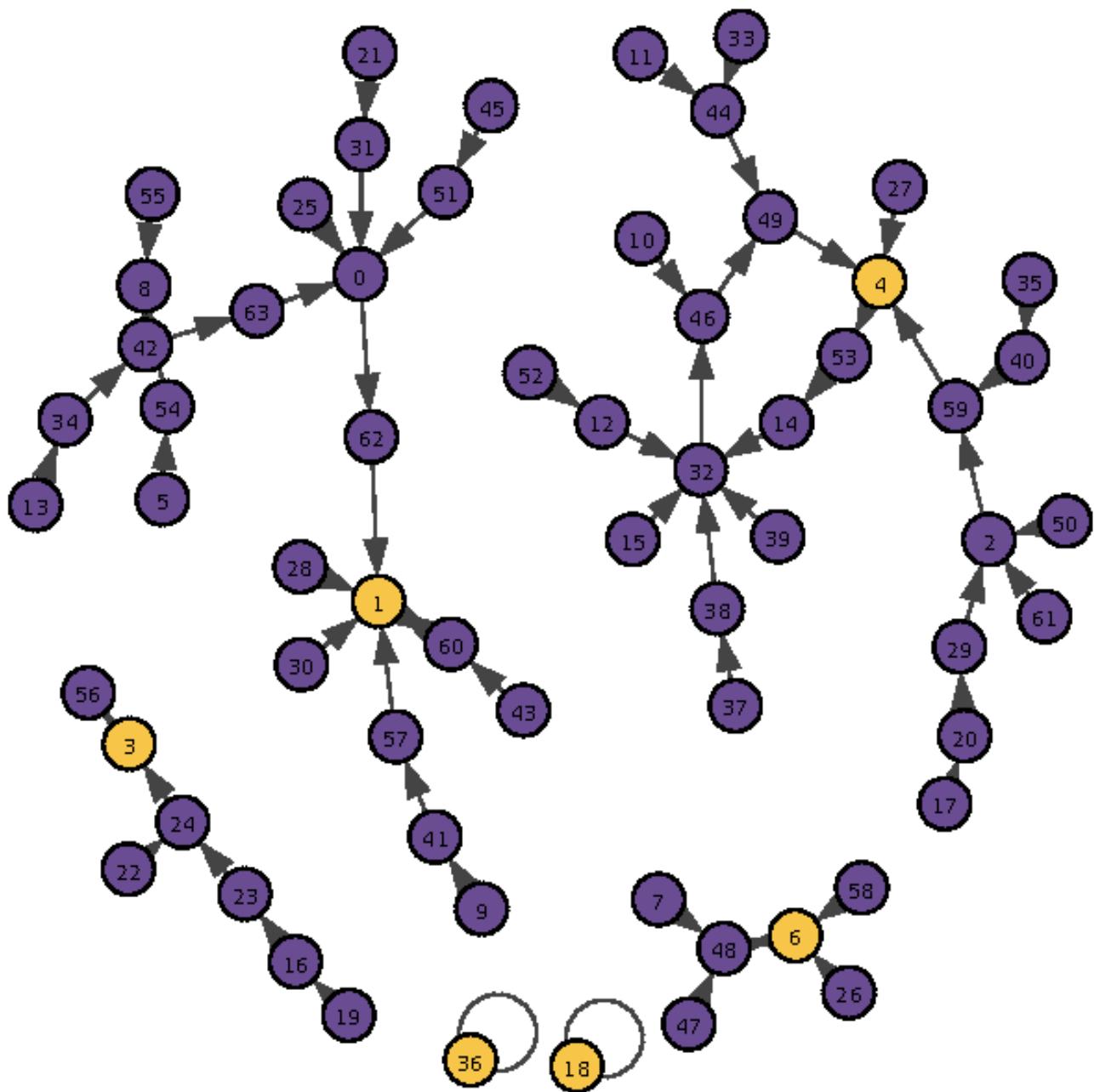


Figure 51: Grafo de los atractores para la regla 37 con $n = 6$

3.1.34 Regla 38

Comportamiento constante en la construcción de un par de árboles de atracción que presentan en sus hojas extremas de las ramas la propiedad de atracción sin ser demasiada fuerte para concentrar un gran número de ancestros cada una.

Junto con este comportamiento constante, se identifica la presencia del estado 0 como uno de los atractores en cada valor de n , y el otro estado corresponde a la configuración que en cantidad decimal es curiosamente siempre la mitad del número de nodos totales evaluados, osea, la mitad de la cardinalidad del universo binario para la potencia n .

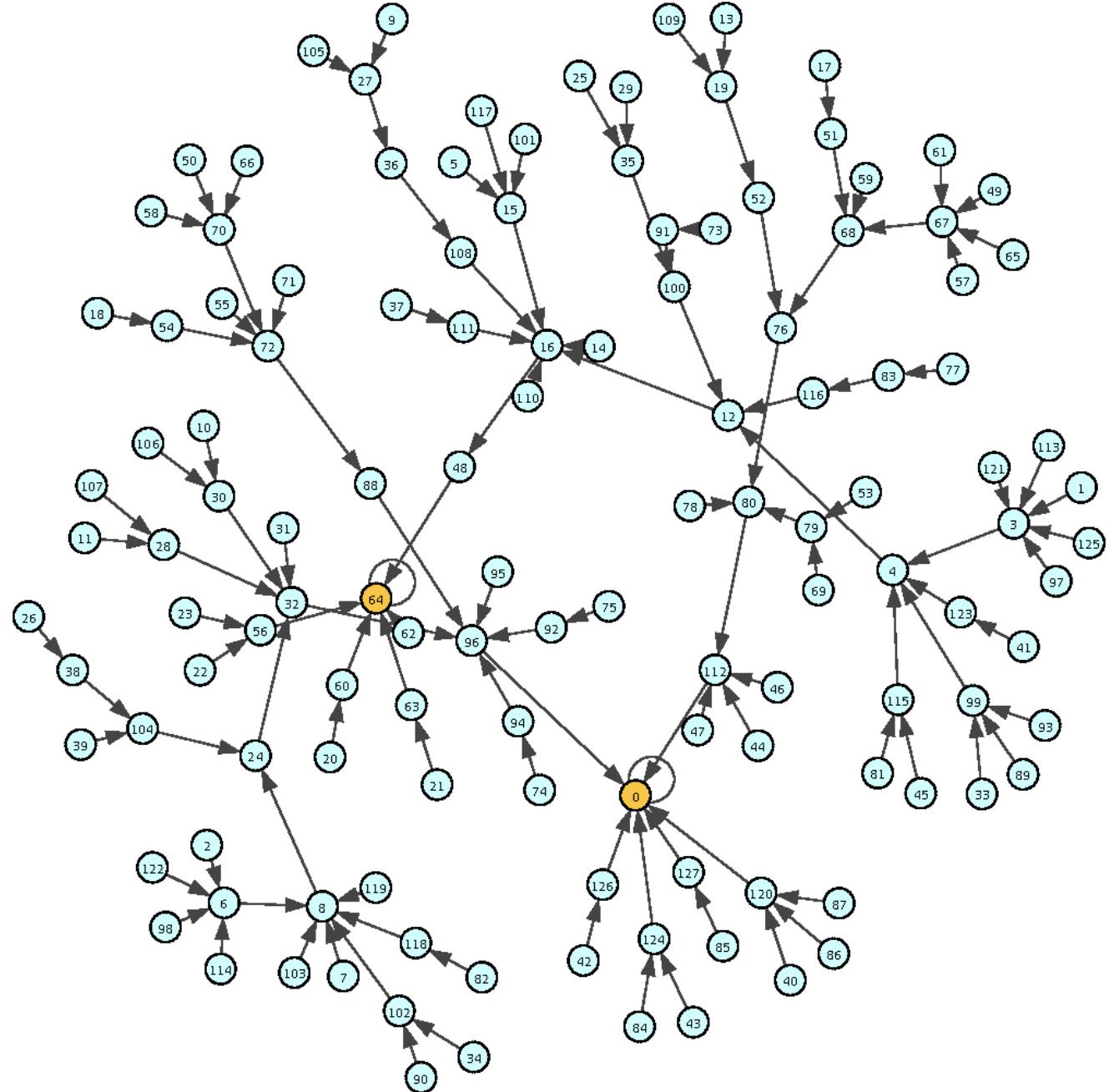


Figure 52: Grafo de los atractores para la regla 38 con $n = 7$

3.1.35 Regla 40

La regla 40 se describe sus campos de atracción al igual que varias otras reglas más donde se construye un árbol de evolución único con el estado 0 como principal atractor. Así también se observa la propiedad de atracción con una particularmente mayor fuerza en nodos que se encuentran a un par de evoluciones de la convergencia del 0, y que disminuye la fuerza de esta atracción a nodos ancestros y predecesores, incluyendo al atractor 0 que no cuenta con esa clara predominancia en la atracción como sucede en otras reglas.

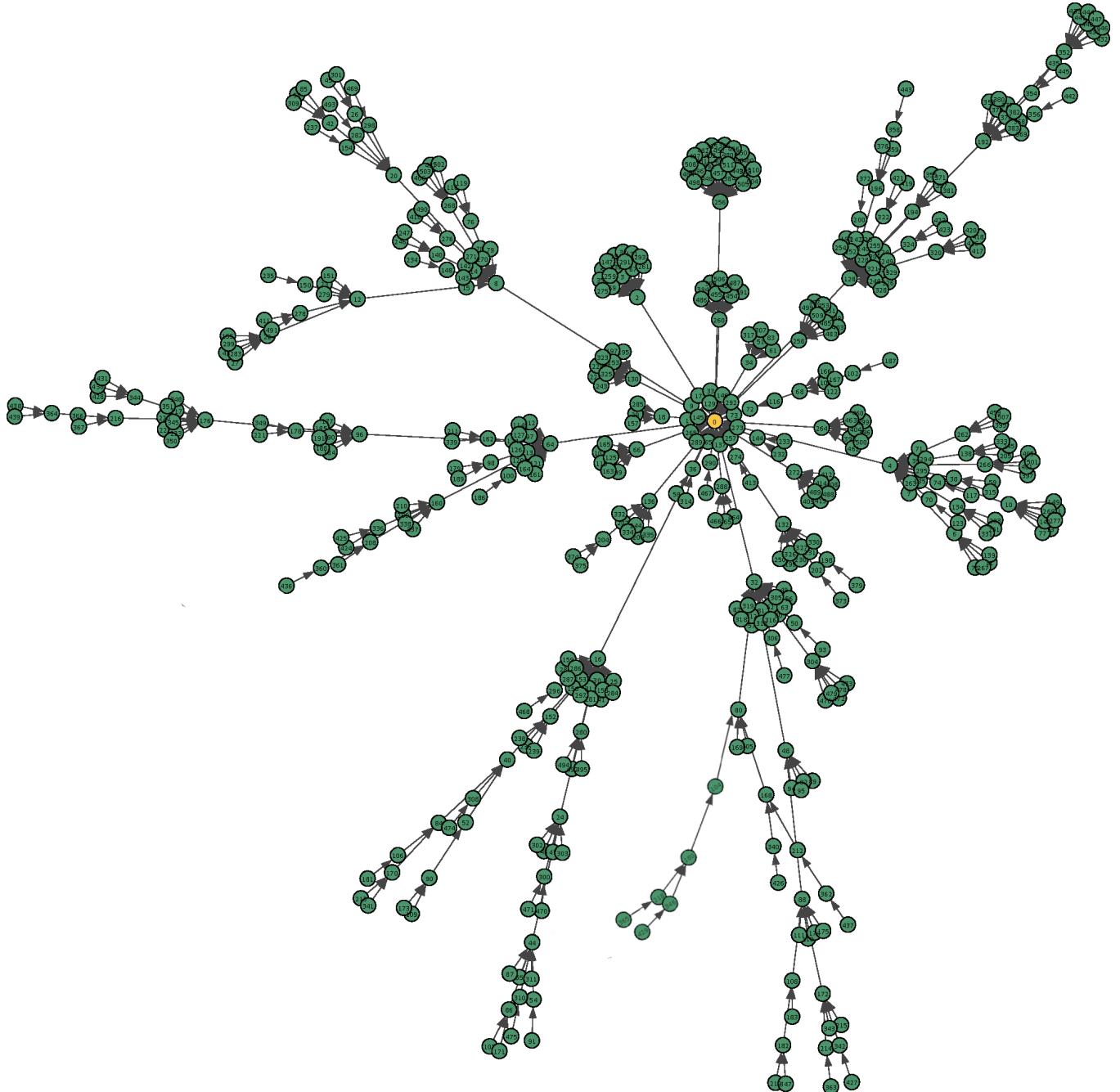


Figure 53: Grafo de los atractores para la regla 40 con $n = 9$

3.1.36 Regla 41

Regla que genera árboles con una distribución extendida conformado por ramas de una gran cantidad de nodos. La generación del número de árboles por potencia de universo binario es creciente pero lenta.

Una cualidad importante a resaltar, es la configuración de los nodos atractores para cada generación, siendo siempre un número par y al menos 1 de los atractores en esa generación, corresponde al valor del número de la potencia del universo(tamaño de células en las configuraciones evaluadas) elevado al cuadrado.

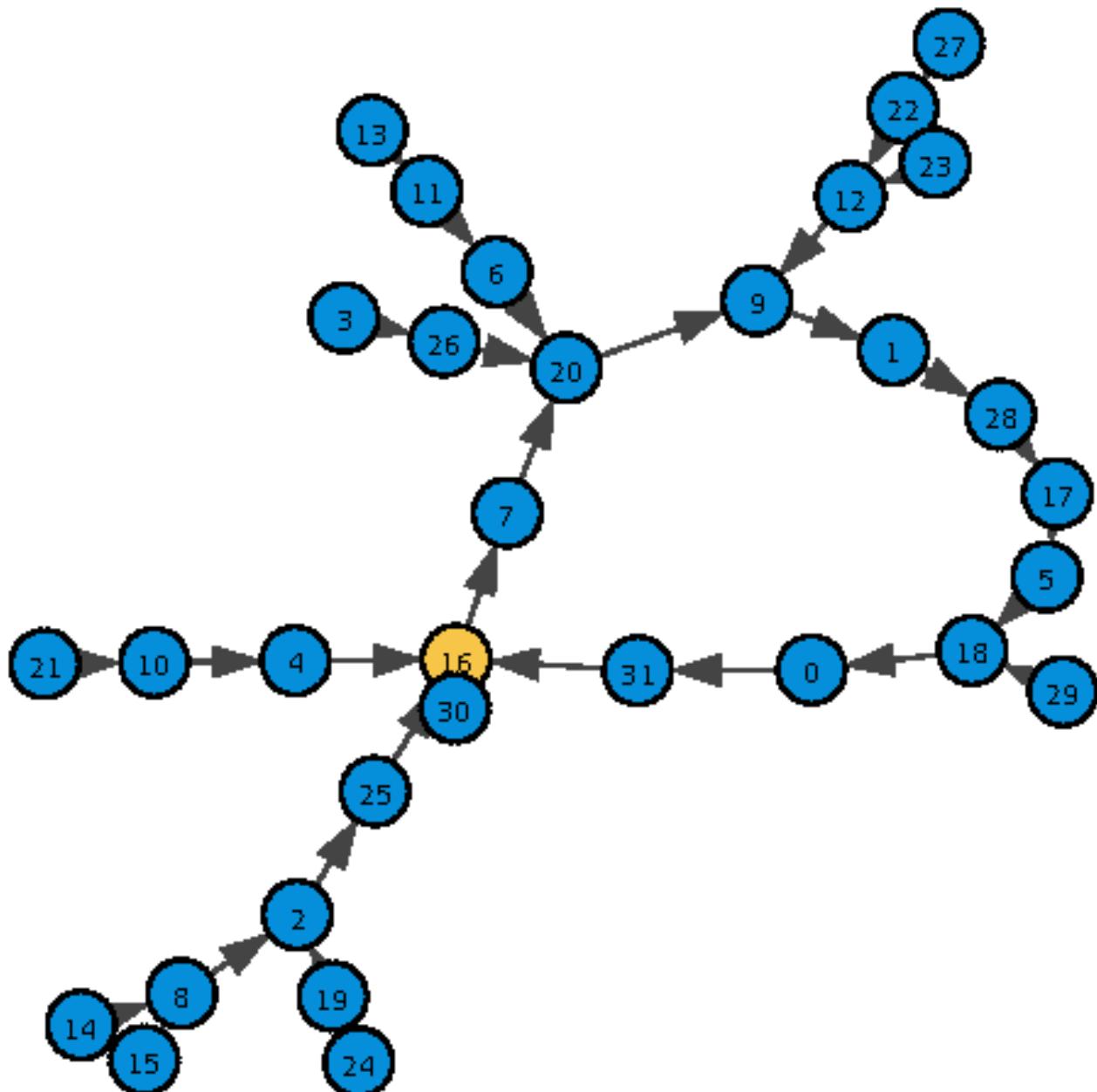


Figure 54: Grafo de los atractores para la regla 41 con $n = 5$

3.1.37 Regla 42

Con comportamiento similar a la regla 34, se genera un árbol único donde el atractor será siempre la configuración 0. Se observan ramificaciones que cuentan con nodos con propiedades atractoras aglomerando una considerable cantidad de ancestros en cada uno.

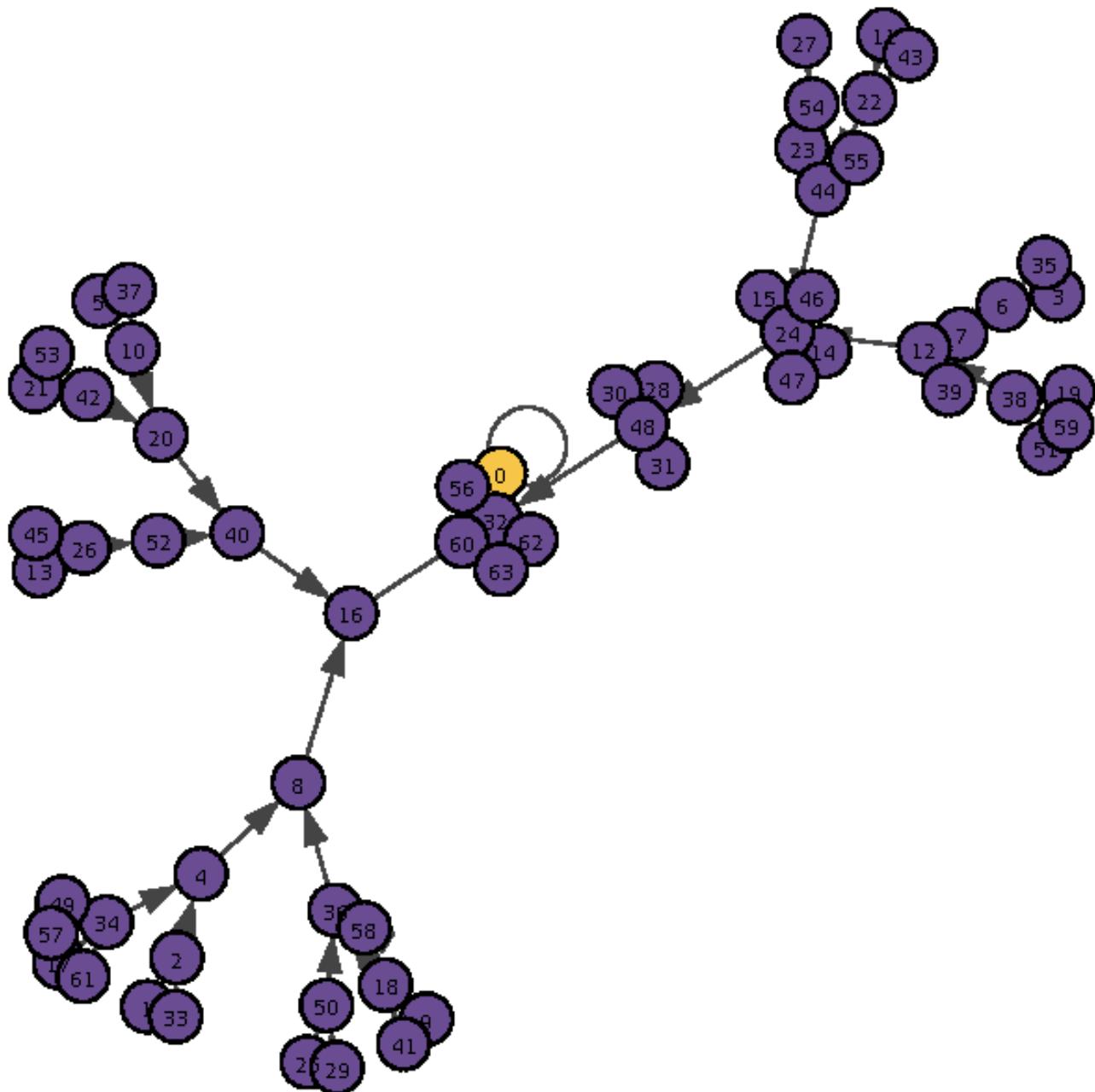


Figure 55: Grafo de los atractores para la regla 42 con $n = 6$

3.1.38 Regla 43

Regla que genera árboles sencillos con una distribución extensa que se traduce en una velocidad más bien lenta para su estabilización. Muestra una tendencia a aumentar de forma lenta pero creciente, el número de árboles que se van creando.

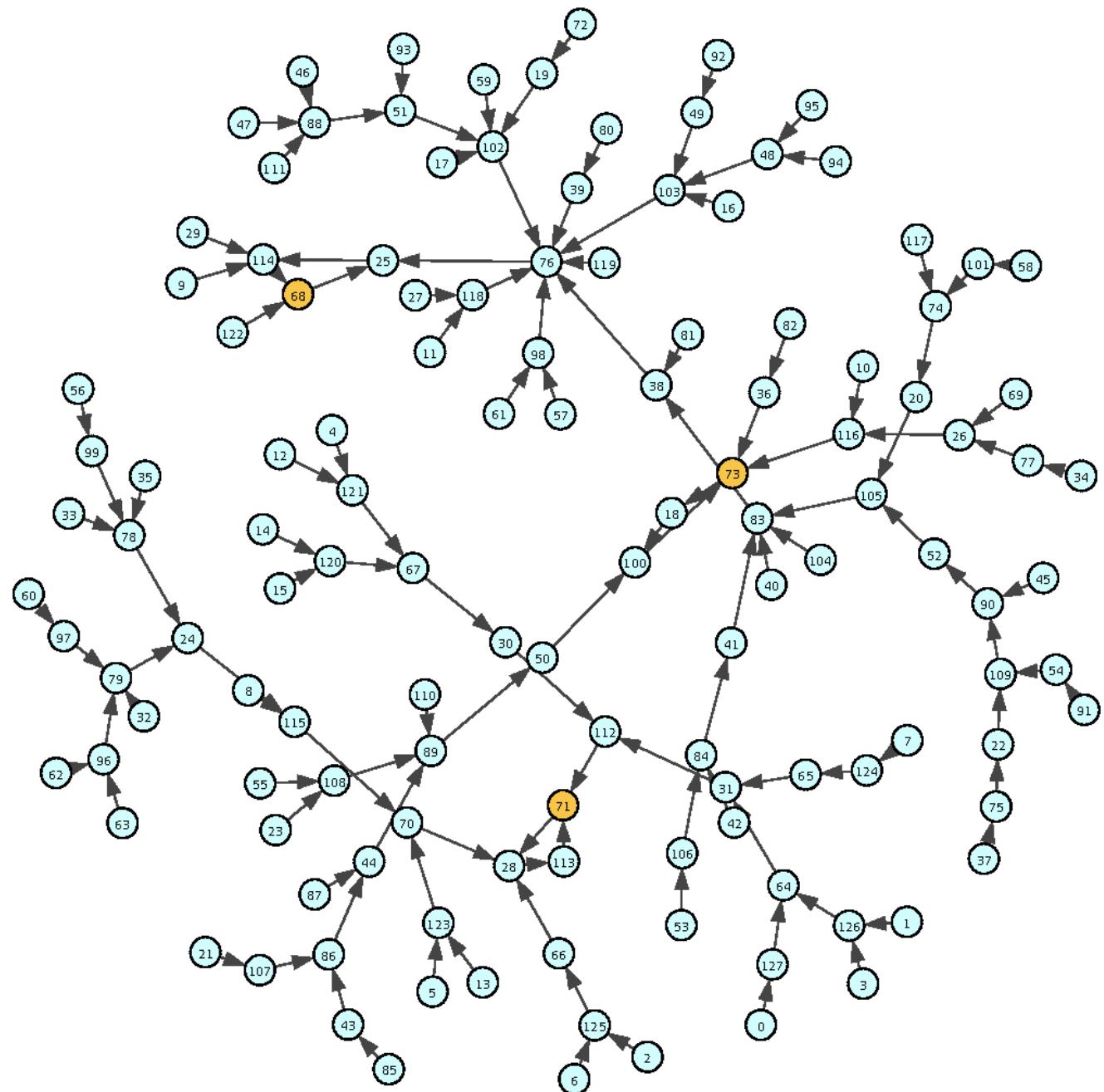


Figure 56: Grafo de los atractores para la regla 43 con $n = 7$

3.1.39 Regla 44

La regla 44 describe a sus campos atractores con una importante cantidad de árboles por generación con un número pequeño de nodos. Las etapas de evoluciones requeridas para la convergencia es también pequeña. Se presentan en tendencia creciente la aparición de atractores marginales.

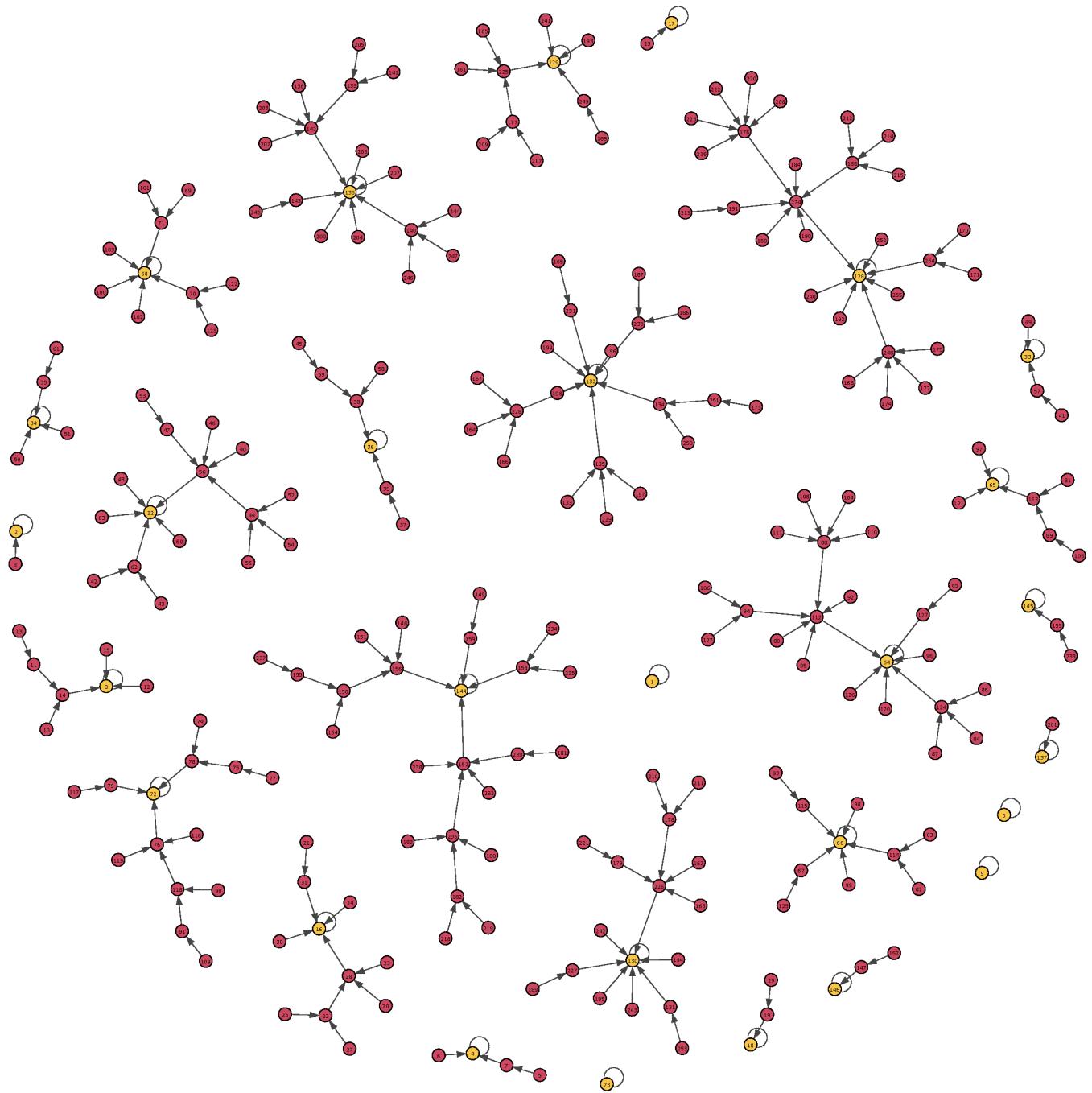


Figure 57: Grafo de los atractores para la regla 44 con $n = 8$

3.1.40 Regla 45

La topología de los árboles generados por la regla 43 son comunes, con amplias ramas sin alta aglomeraciones de nodos en los extremos y por lo tanto una velocidad de estabilización lenta.

Como particularidad en esta evolución se observa un comportamiento recurrente en la mayoría de las evoluciones (con excepción de la 9), donde al menos 1 atractor descrito en la generación menor inmediata se describe también en la siguiente.

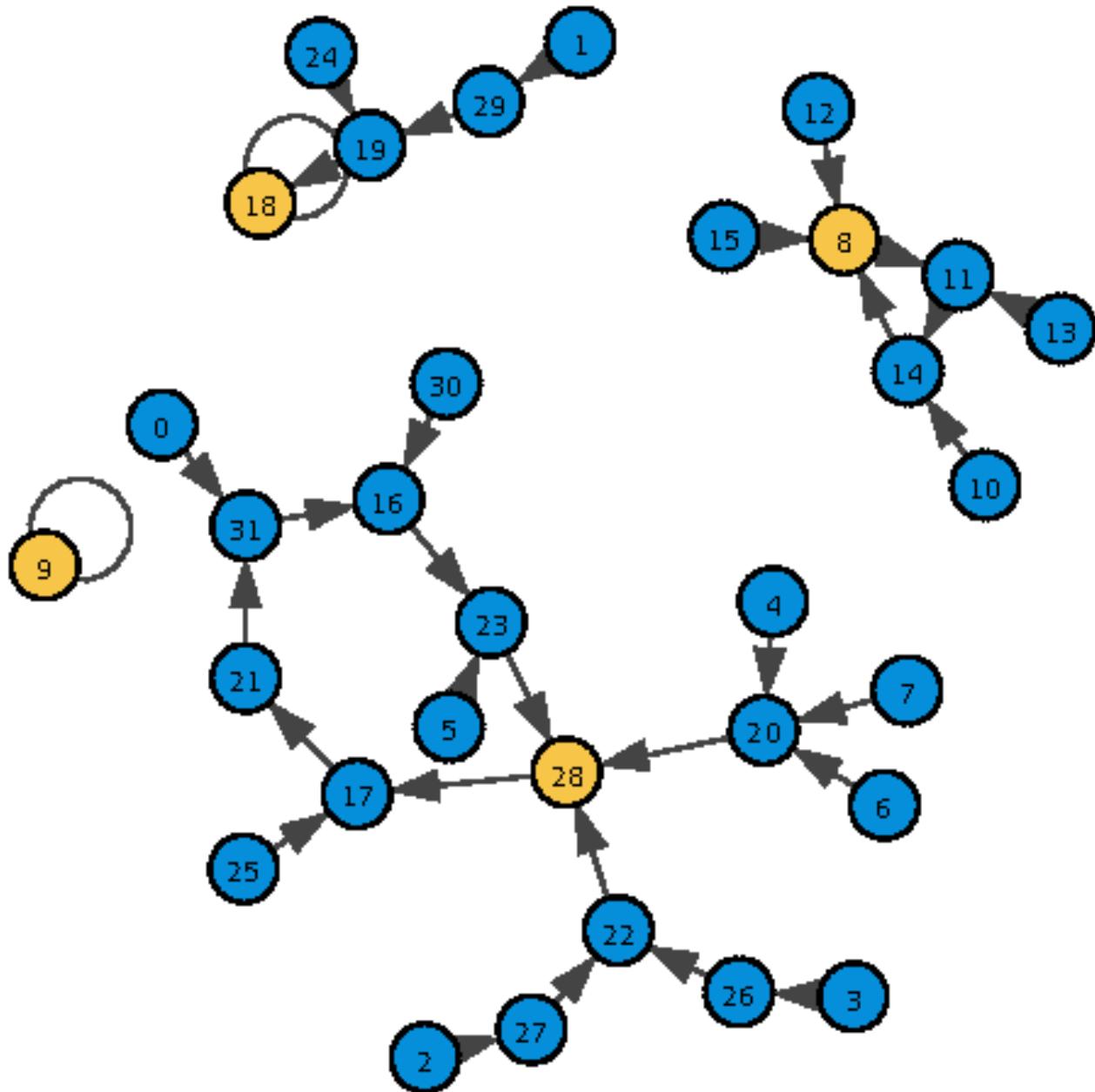


Figure 58: Grafo de los atractores para la regla 45 con $n = 9$

3.1.41 Regla 46

Generador del atractor marginal 0 y un árbol único de evolución con un conjunto de nodos pertenecientes a las ramas que muestran cualidades de atracción aglomerando una cantidad importante de nodos del edén.

Al igual que se ha descrito con otras reglas, el atractor del árbol de atracción que se aprecia, es la potencia de 2 resultante de elevar 2^n , donde n es el número de células utilizadas en la evolución.

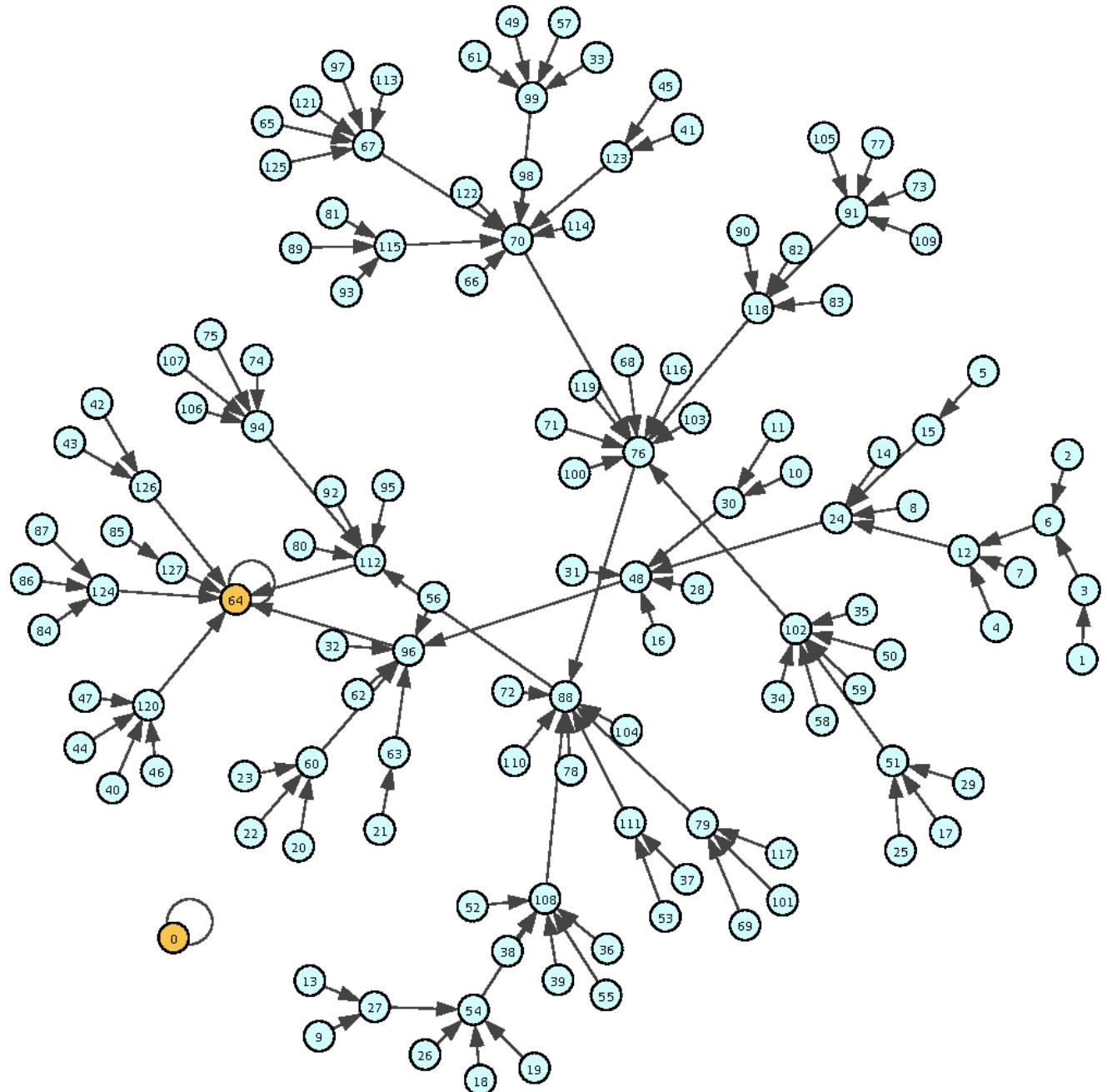


Figure 59: Grafo de los atractores para la regla 46 con $n = 9$

3.1.42 Regla 50

Los árboles creados para esta regla describen una tendencia creciente a medida que aumenta n , en el número de árboles generados. Estos árboles tienen una topología bien distribuida por lo que no existen cualidades atractoras en otros nodos que no se hayan encontrado atractores.

Se destaca la construcción de los árboles, observándose en una misma generación, aquellos conformados de 2 nodos y que van creciendo para alcanzar el árbol principal que lo compone la mayoría de nodos.

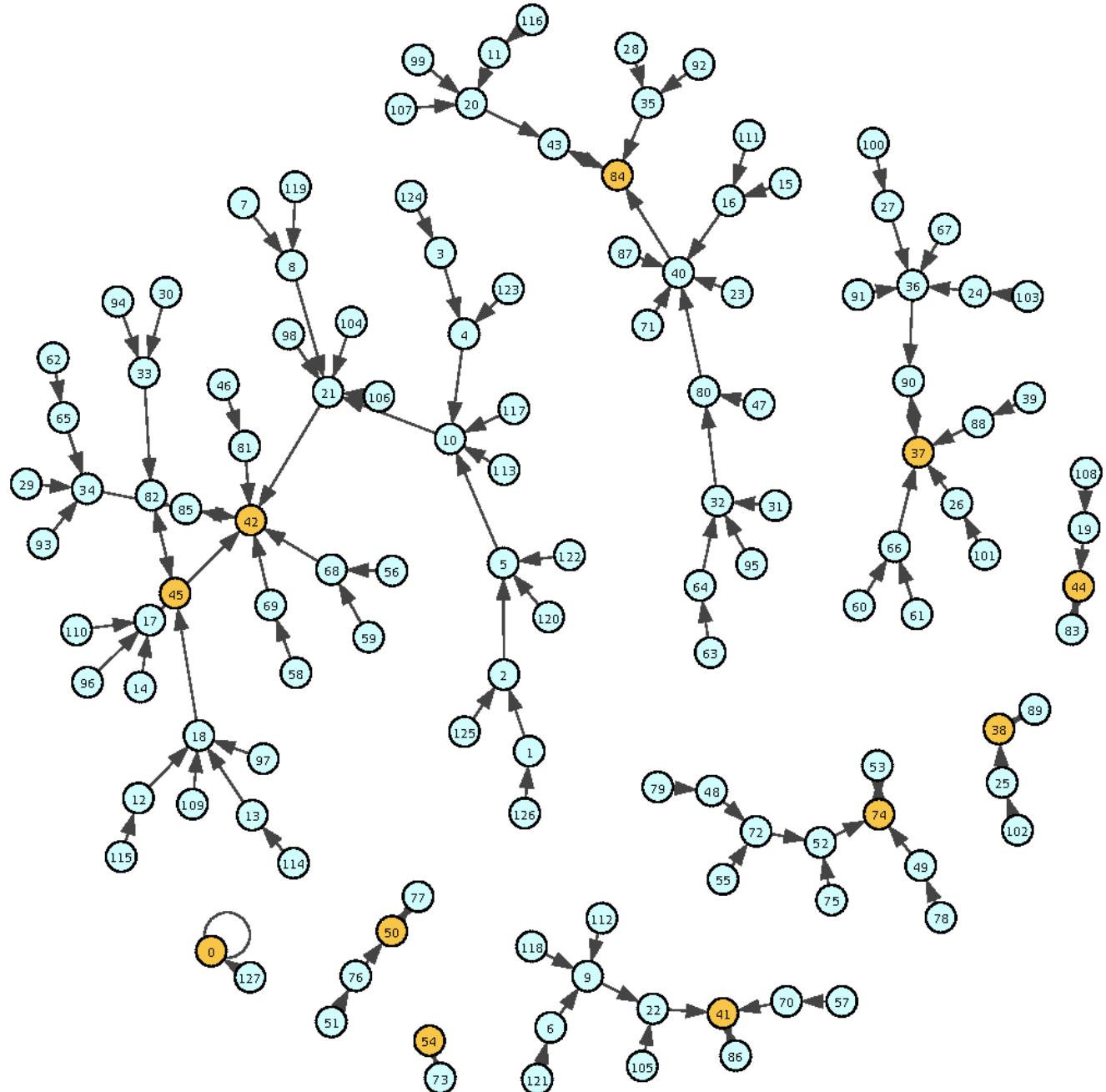


Figure 60: Grafo de los atractores para la regla 50 con $n = 7$

3.1.43 Regla 51

La regla 51 muestra la construcción de campos de atracción exclusivamente compuestos por 2 nodos, esto evidentemente va generar un gran conjunto de atractores en cada una de las generaciones, pero también aumenta la velocidad de convergencia a tan solo una fase de evolución.

Las configuraciones atractores serán siempre los nodos que van desde el 0 hasta la mitad del conjunto de nodos, siendo la segunda mitad siempre los nodos raíz.

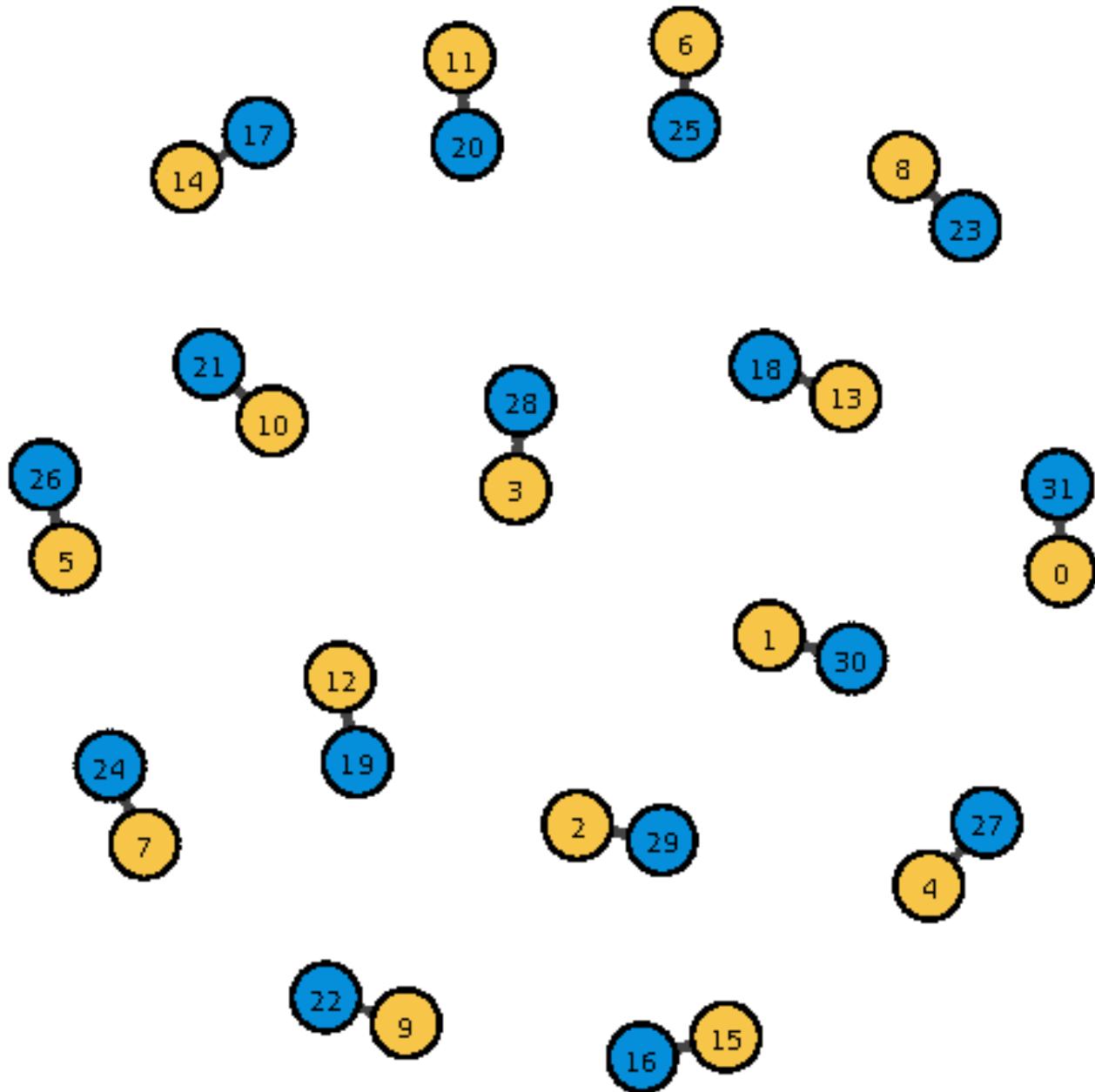


Figure 61: Grafo de los atractores para la regla 51 con $n = 5$

3.1.44 Regla 54

Esta regla fabrica árboles con 3 diferencias topológicas entre ellos. Siempre presente el atractor 0, se caracteriza la conformación de este árbol con un único ancestro el cual muestra propiedades de atracción importantes, aglomerando a su alrededor nodos del edén.

El segundo tipo de árbol que se encuentra es de extensión pequeña pero que forma un característico ciclo de 4 nodos con el atractor, y de las que pueden surgir ramas que directamente se unen a la configuración atractora sin deformar su ciclo.

Finalmente el tercer tipo de árbol es uno común, de gran extensión, con ramas de nodos secuenciales que describen una convergencia lenta.

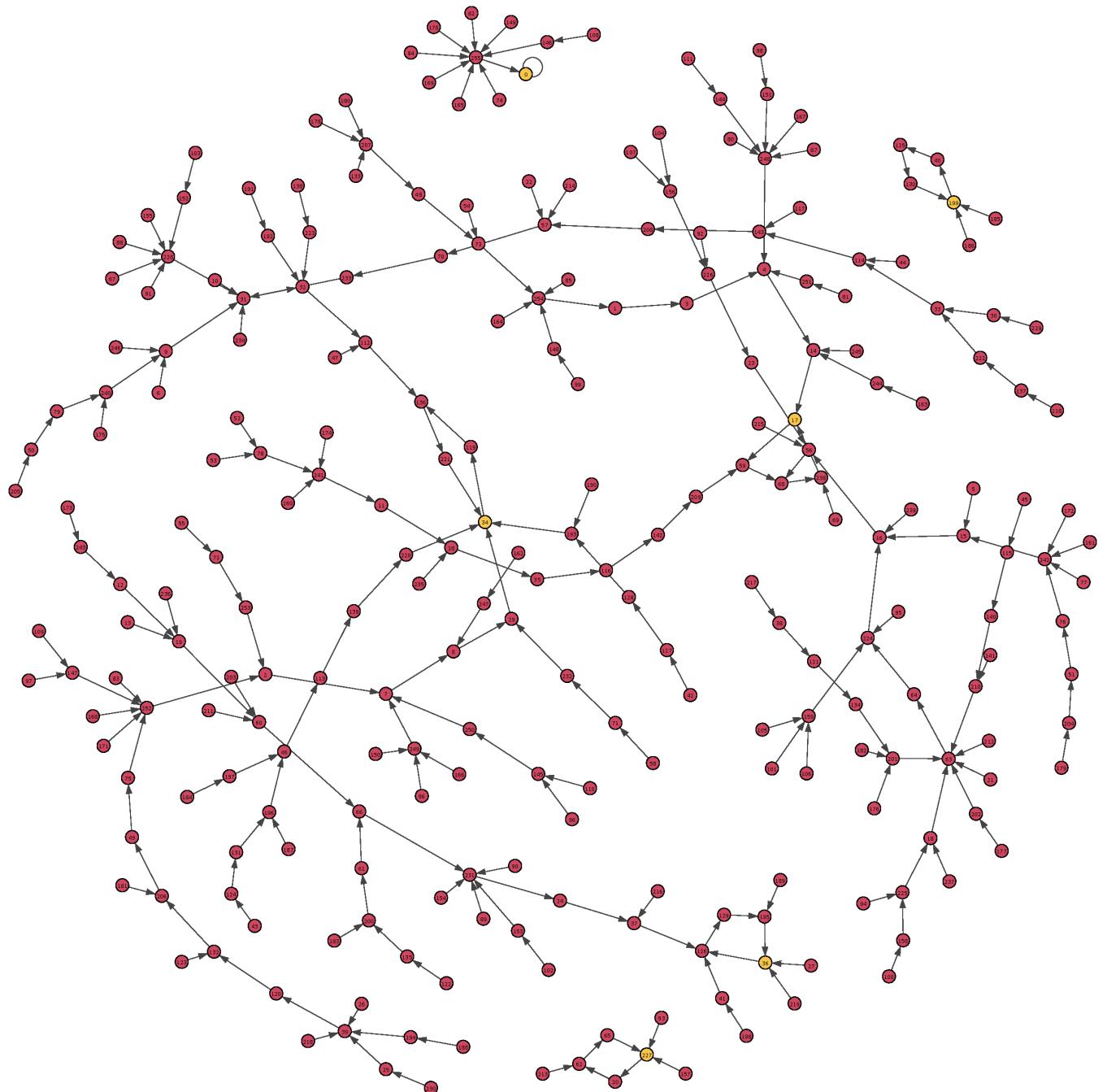


Figure 62: Grafo de los atractores para la regla 54 con $n = 8$

3.1.45 Regla 56

Generador de un árbol único con atractor al nodo 0, tiene como característica que sus nodos en los extremos de las ramas presentan propiedades atractoras, de forma que una cantidad de nodos del edén son ancestros de 1 solo nodo.

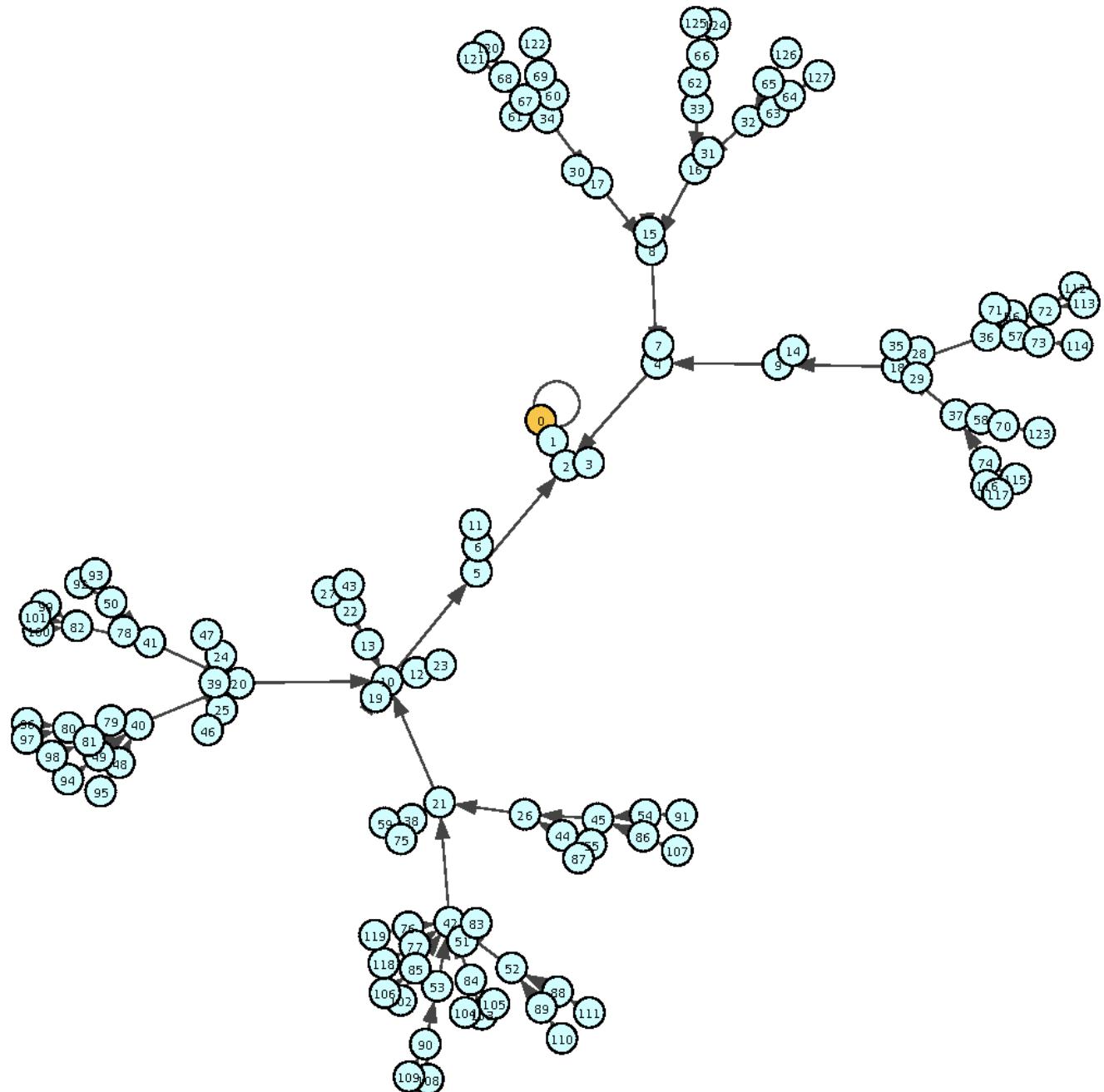


Figure 63: Grafo del atractor para la regla 56 con $n = 7$

3.1.46 Regla 57

Los árboles de atracción de la regla 57 se describen con árboles de mediana extensión con varios nodos secuenciales conformando sus ramas, pero no siendo demasiado grandes pues el conjunto de nodos se reparten entre varios árboles. La regla muestra una tendencia creciente al número de árboles que se generan.

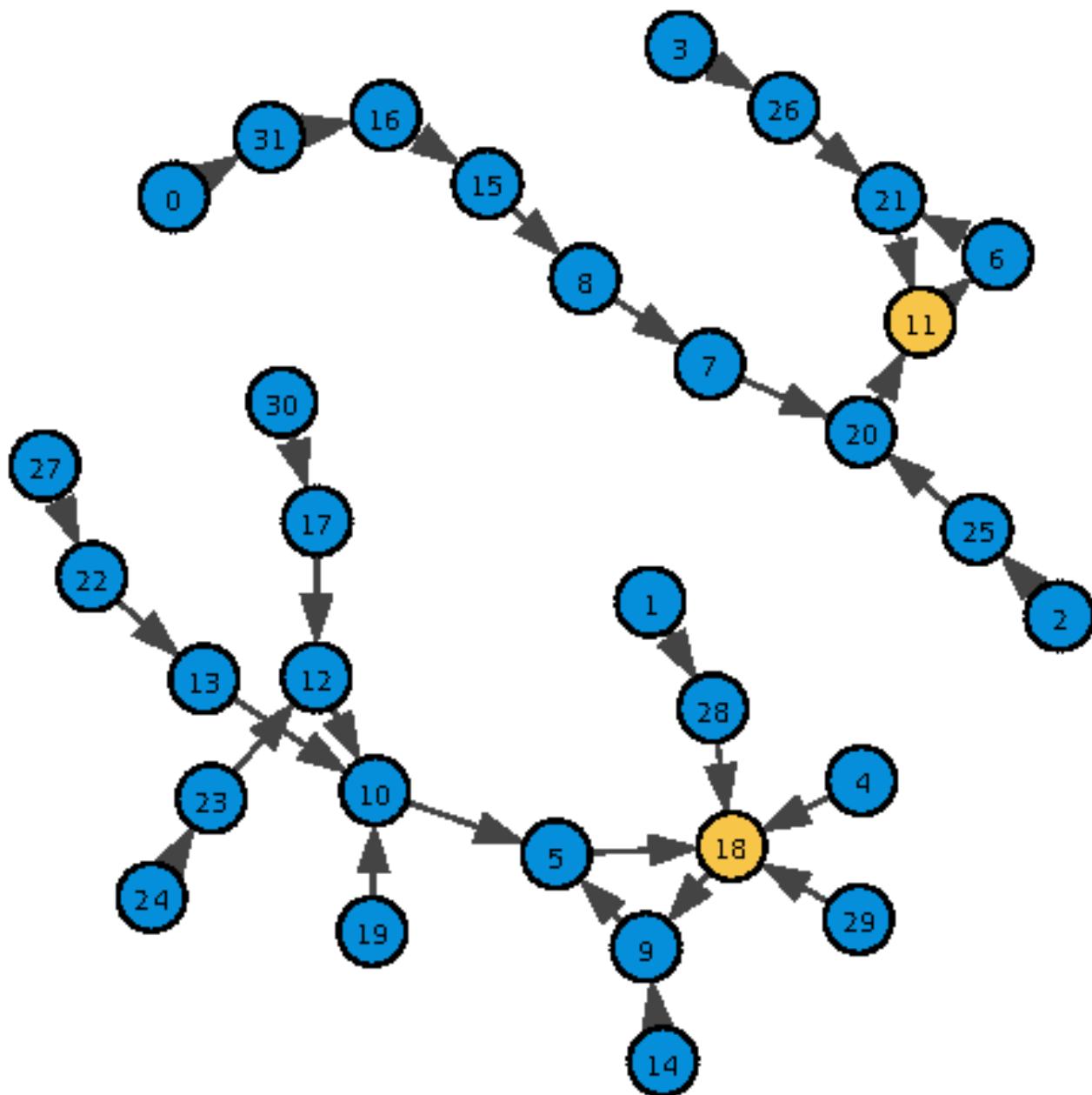


Figure 64: Grafo de los atractores para la regla 57 con $n = 5$

3.1.47 Regla 58

Regla fabricante de 2 nodos atractores por generación: El 0 como atractor marginal(presente únicamente en potencias n impares) y un árbol de evoluciones con una característica rama principal al atractor que se compone de varios nodos en secuencia que tiene la particularidad de generar ramas cada uno, siendo ramas de no una extensión tan grande pero que si cohesiona a varios nodos a su alrededor.

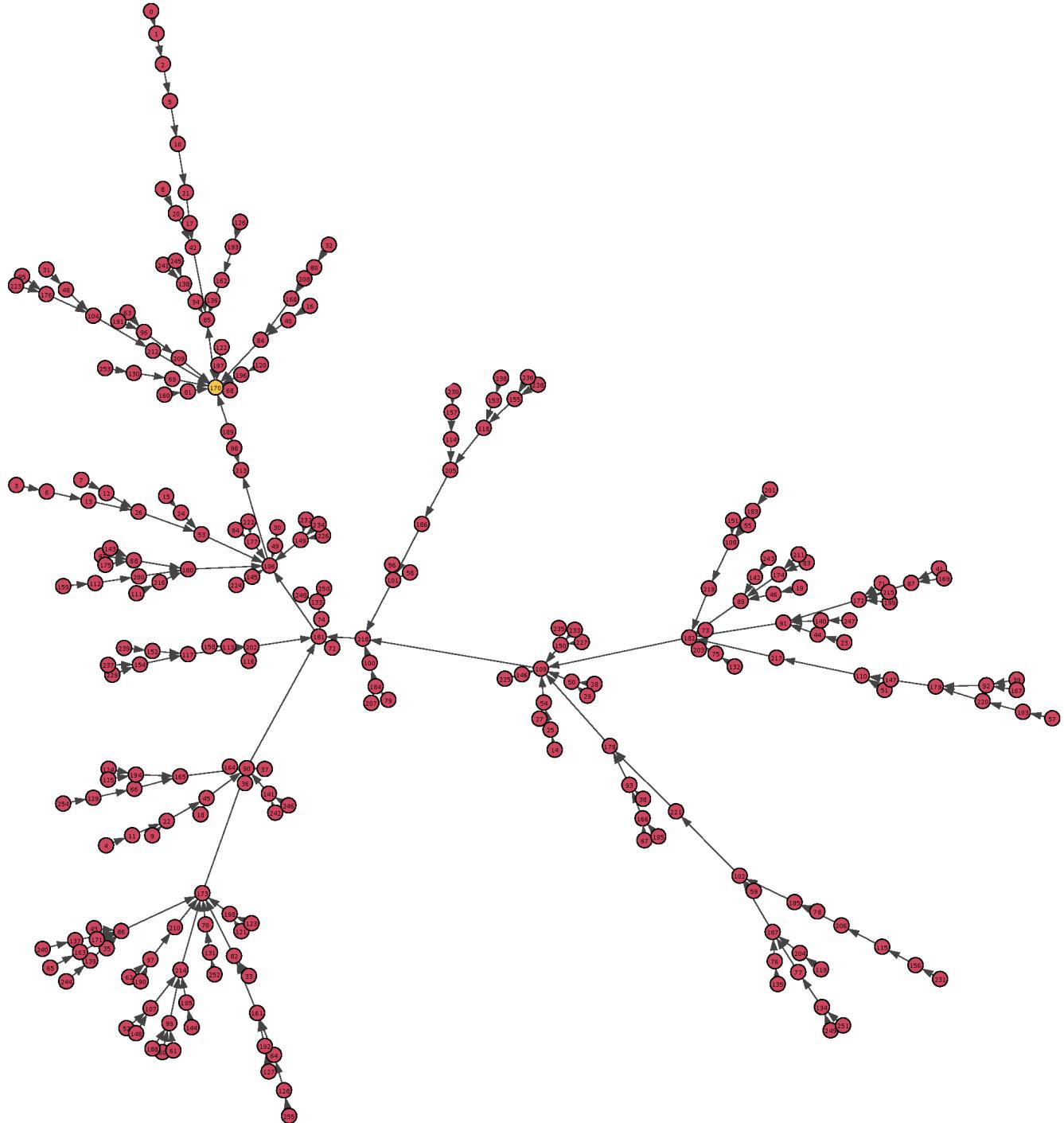


Figure 65: Grafo de los atractores para la regla 58 con $n = 8$

3.1.48 Regla 60

La generación de los campos atractores de la regla 60 es muy particular en el sentido que sus generaciones consiste de únicamente ciclos simples, no hay construcción de árboles con ramas extensas, aunque se advierte la presencia de los nodos 0 y 1 como atractores marginales.

Particularmente la construcción de los ciclos se mantiene entre diferentes generaciones. Implicando que con generaciones mayores tan solo se agregan nuevos ciclos con los nodos no explorados anteriormente pero las estructuras construidas con valores de n menores no se destruyen ni se inmutan.

Esta generación es particularmente especial pues describe una regla en el que los ECAs son capaces de de generar cada una de las configuraciones posibles existentes, esto por la ausencia de los nodos edén.

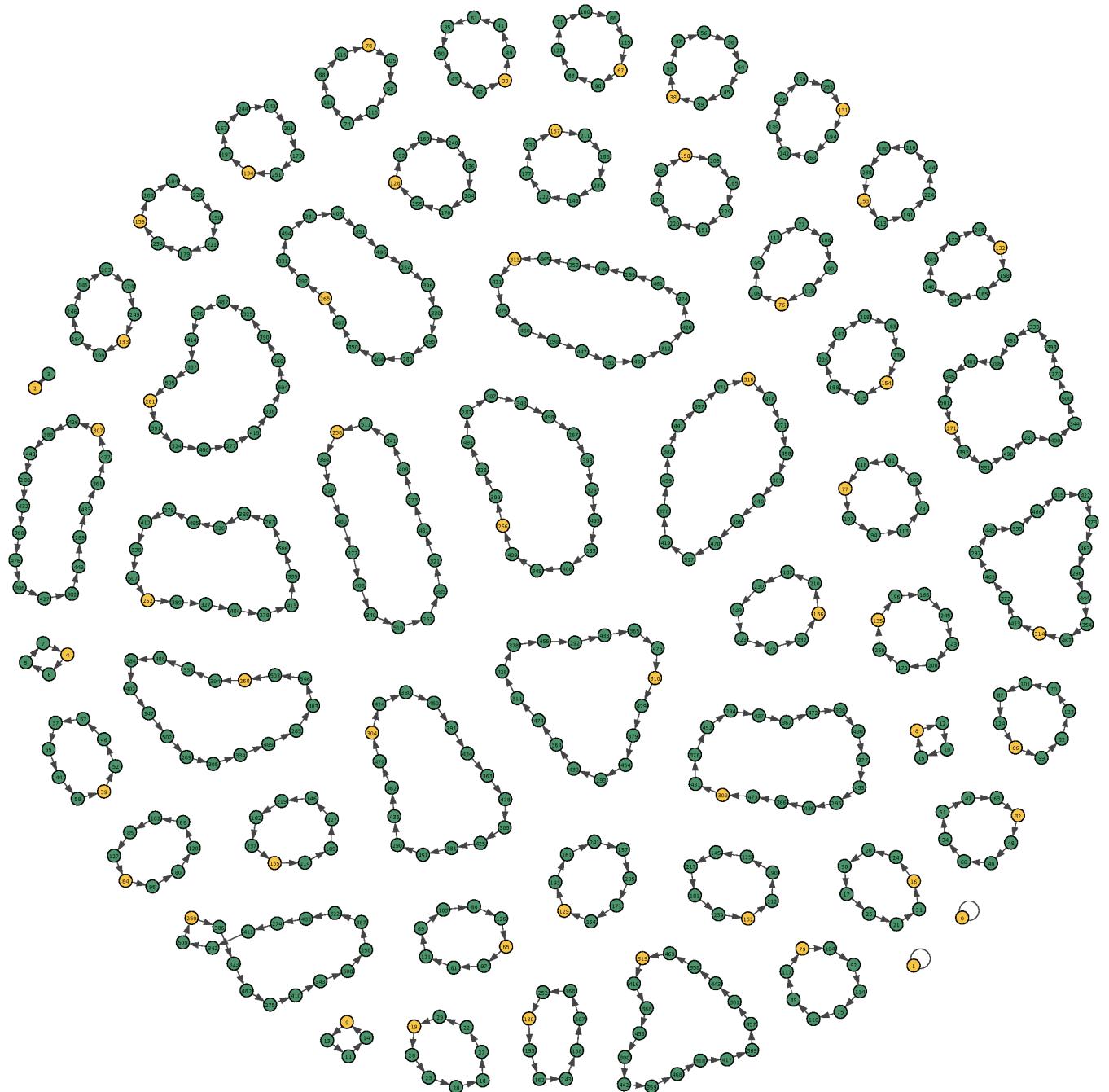


Figure 66: Grafo de los atractores para la regla 60 con $n = 9$

3.1.49 Regla 62

Regla constructora de árboles de evolución con una tendencia creciente a incrementar el número de árboles que genera así como también aumenta el valor de n .

Se observa la constante presencia del nodo 0 como atractor marginal en todas las generaciones.

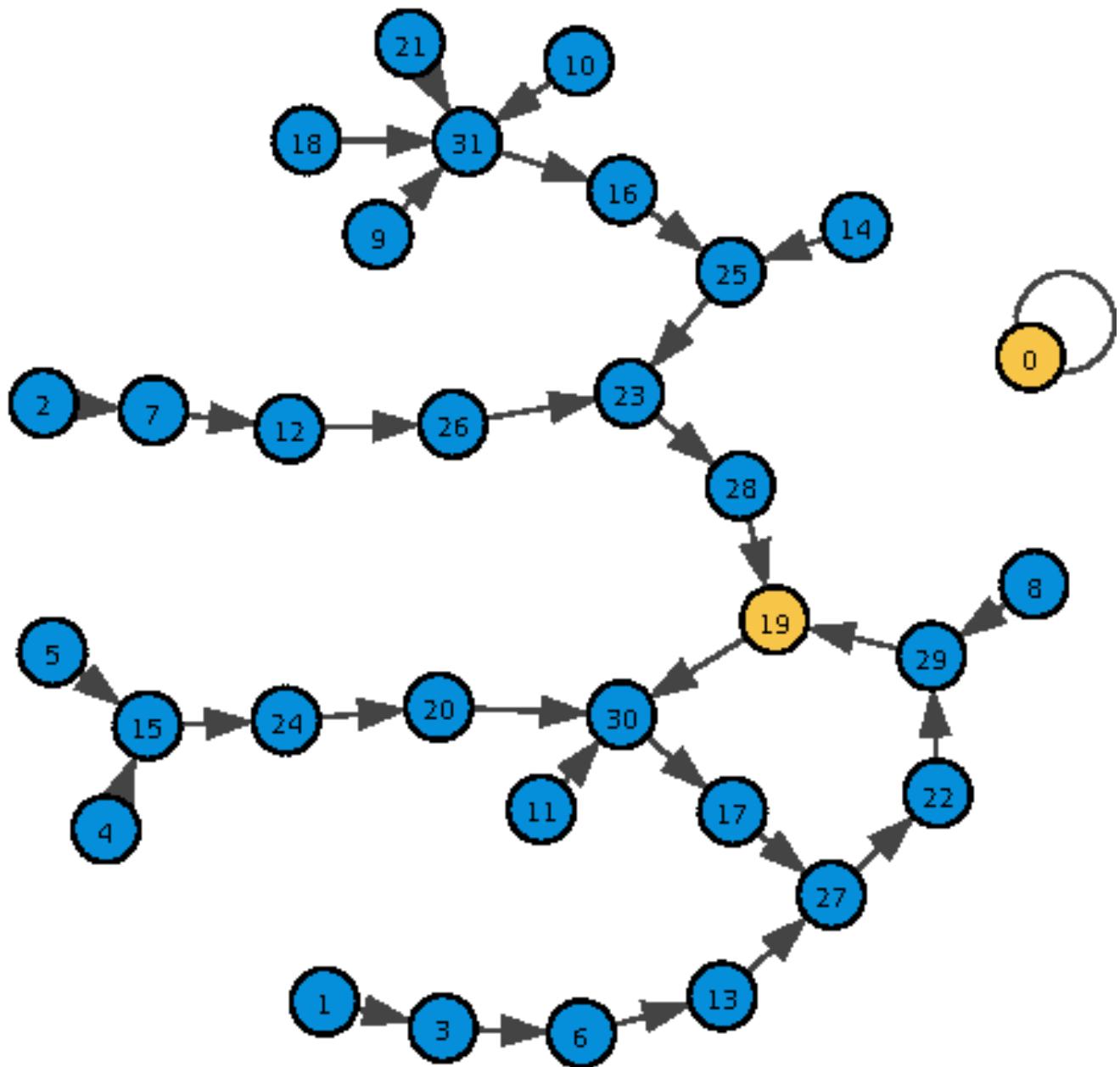


Figure 67: Grafo de los atractores para la regla 62 con $n = 5$

3.1.50 Regla 72

Regla productora de árboles de evolución con pocas fases de evolución requeridas para la convergencia. Se observa una importante capacidad de atracción para nodos extremos en las ramas de cada árbol, y sin embargo destacando de entre todas las estructuras el árbol del atractor 0, siendo evidentemente el más importante en cuanto a estabilización refiere por el número de configuraciones que se unen a este.

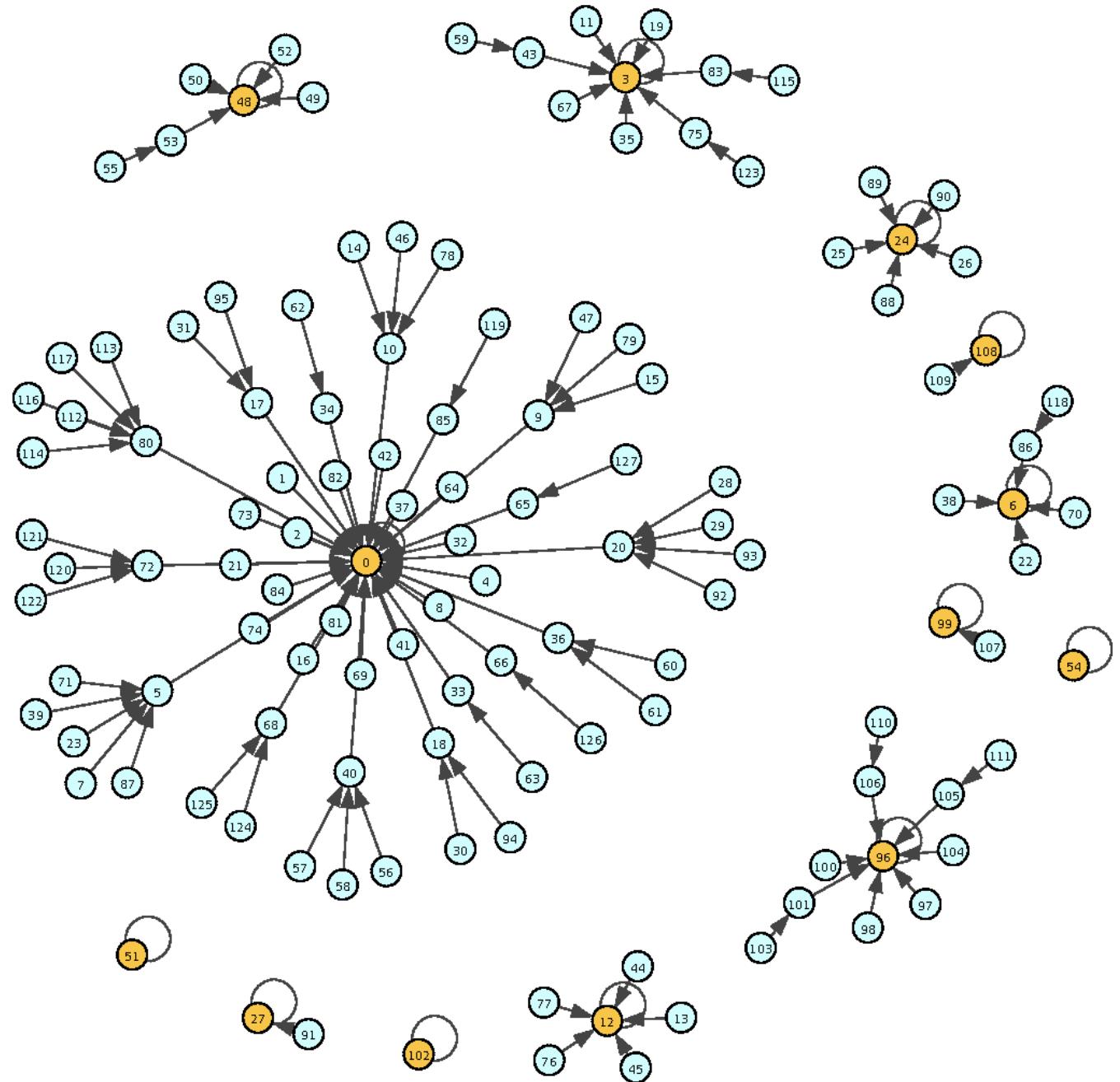


Figure 68: Grafo de los atractores para la regla 72 con $n = 7$

3.1.51 Regla 73

Regla constructora de árboles atractores en gran número por cada generación. Estos varian en extensión y nodos que los conforman, habiendo uno con una gran número de nodos, el atractor 0, y los demás paulatinamente se muestran con una extensión menor.

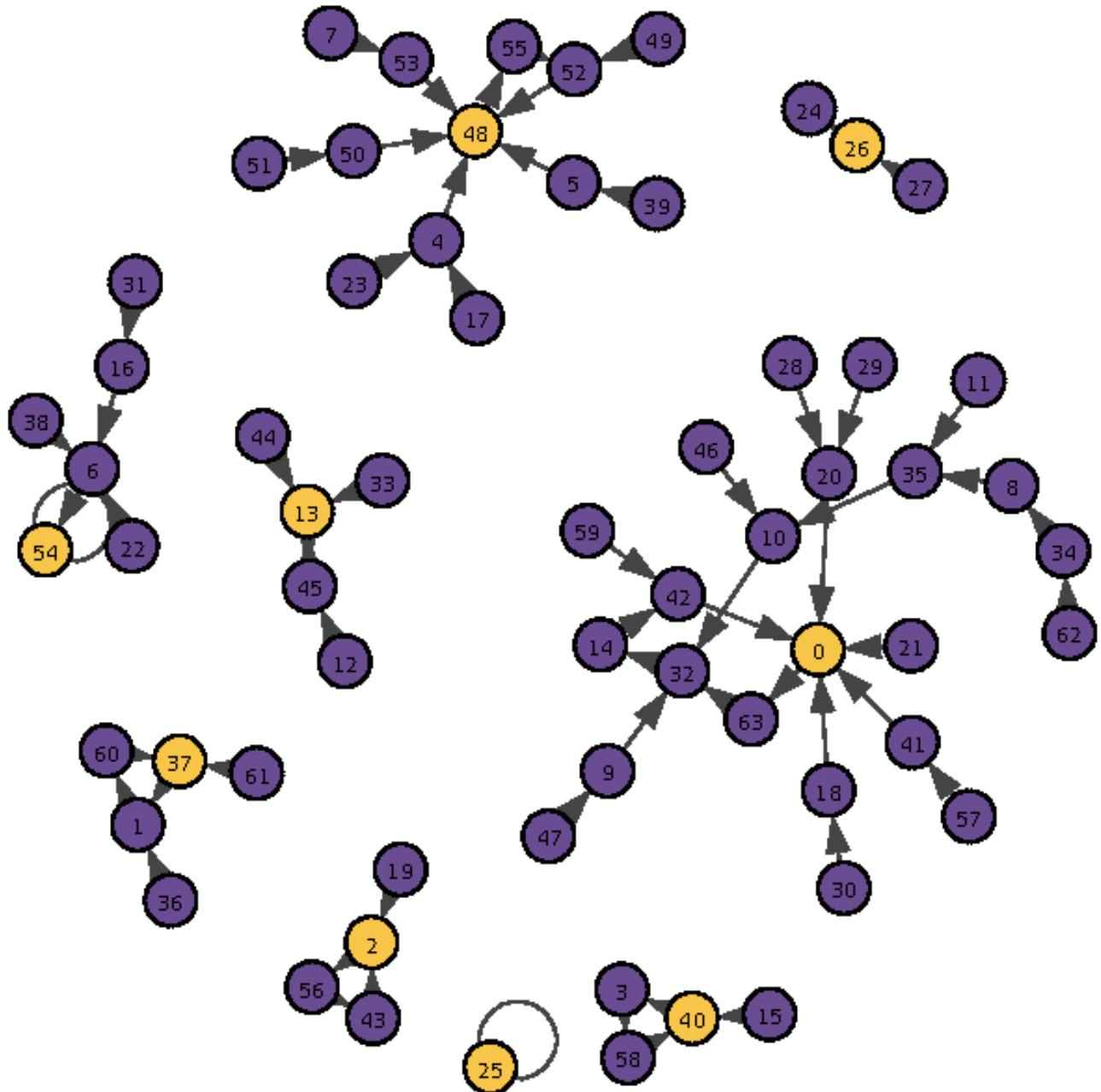


Figure 69: Grafo de los atractores para la regla 73 con $n = 6$

3.1.52 Regla 74

Fabricante de árboles con una gran extensión en su topología, un importante número de nodos consecutivos crean las ramas de los árboles, pero siempre destacando el árbol del atractor 0, siendo evidentemente el más importante de los generados.

Se advierte la presencia de 1 atractor marginal, que aunque desaparece en algunas generaciones, conserva la tendencia creciente como múltiplo de 3.

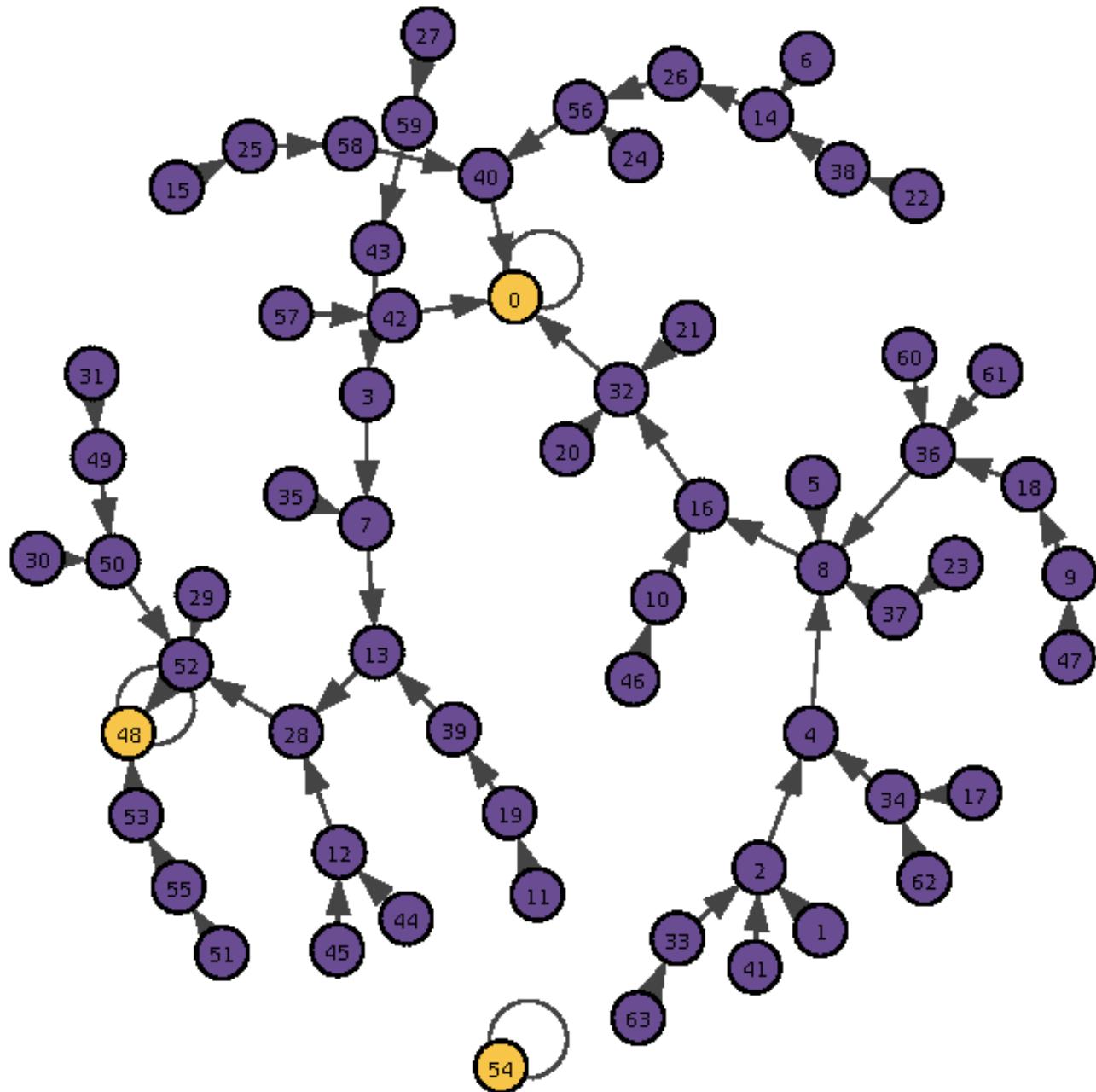


Figure 70: Grafo de los atractores para la regla 74 con $n = 6$

3.1.53 Regla 76

Regla constructora de campos de atracción con mínimas fases de evolución a la estabilización. La distribución de pocos nodos por atractor exige de la generación la creación de una impresionante cantidad de atractores, los cuales sin embargo, es su mayoría son atractores marginales.

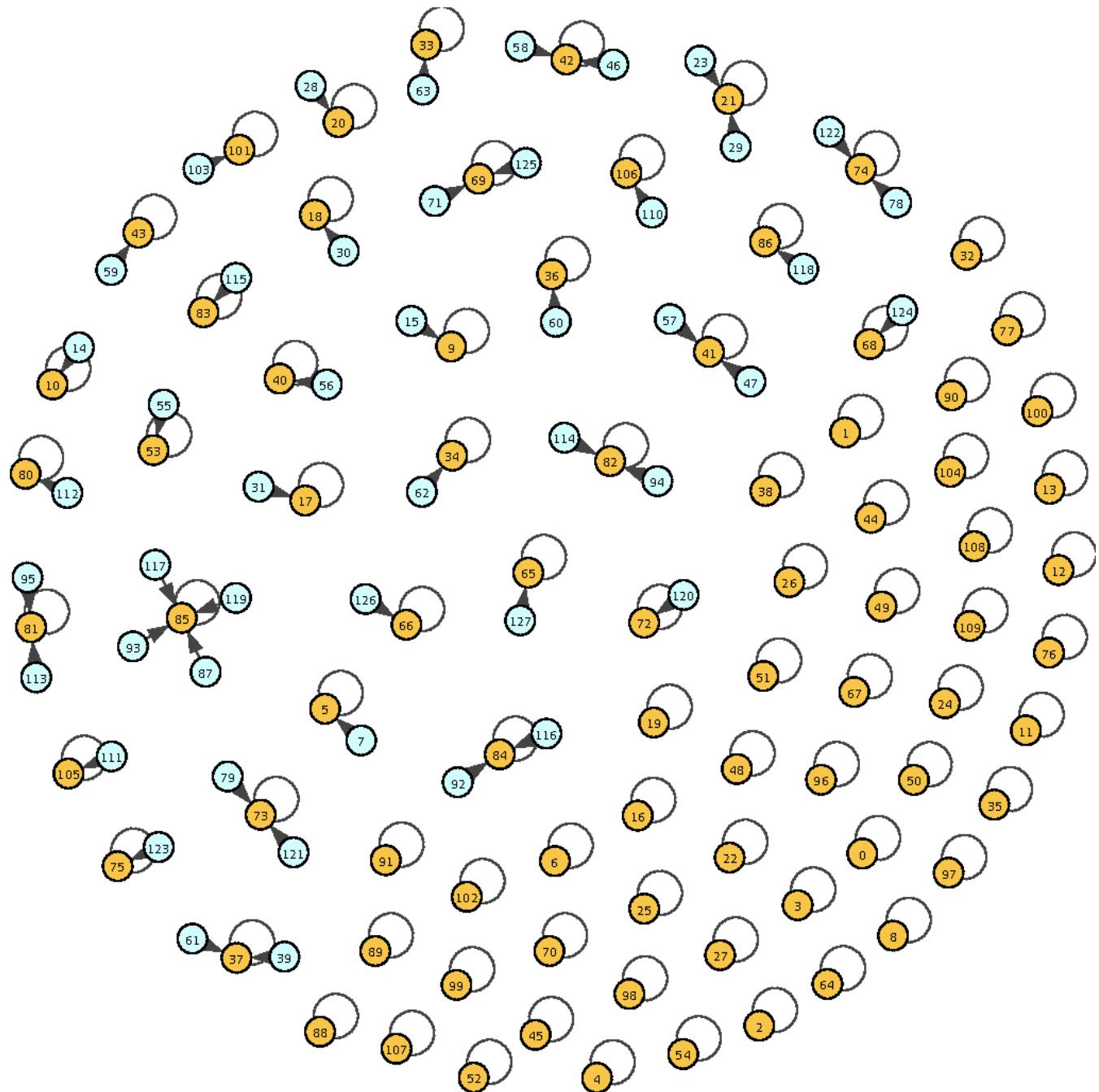


Figure 71: Grafo de los atractores para la regla 76 con $n = 7$

3.1.54 Regla 77

Los campos de atracción para la regla 77 se describen con un conjunto de atractores principales, los cuales cuenta con un gran número de nodos conformando su árbol de evolución.

Mientras los demás atractores con un número disminuido de ancestros, siguen mostrando la cualidad de atracción aglomerando hojas del edén.

De forma general se observa una poca consecución de nodos en las ramas, por lo que la velocidad de convergencia es bastante rápida, tomando en cuenta también que la presencia de nodos marginales implica que no se requieren fases de evolución cuando se utilizan tales configuraciones.

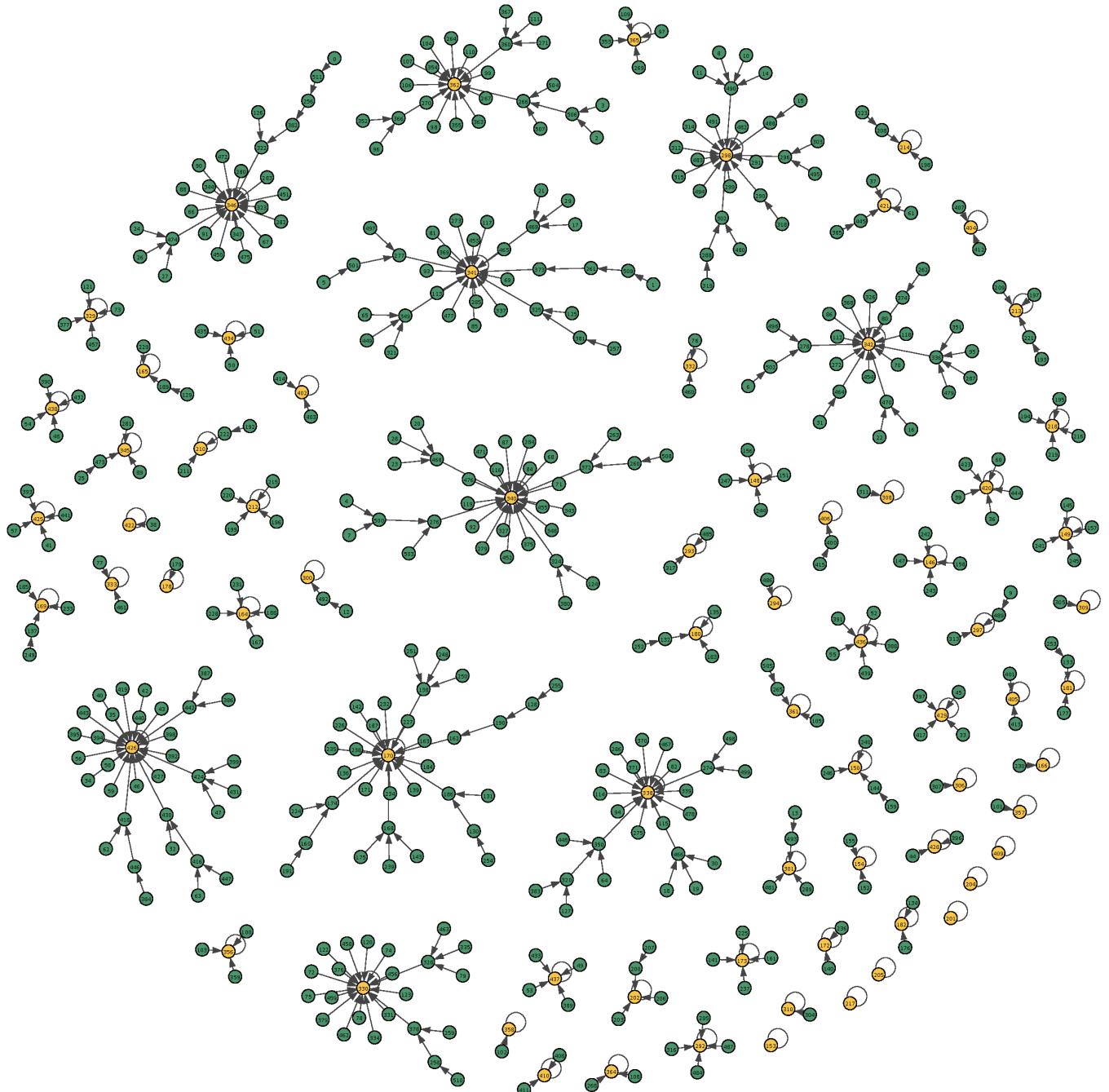


Figure 72: Grafo de los atractores para la regla 77 con $n = 9$

3.1.55 Regla 78

Regla constructora de árboles de evolución con ramas construidas por nodos consecutivos, no existen aglomeraciones importantes de ancestros en un solo nodo. También se observa una tendencia creciente de fabricación de un gran número de árboles por generación.

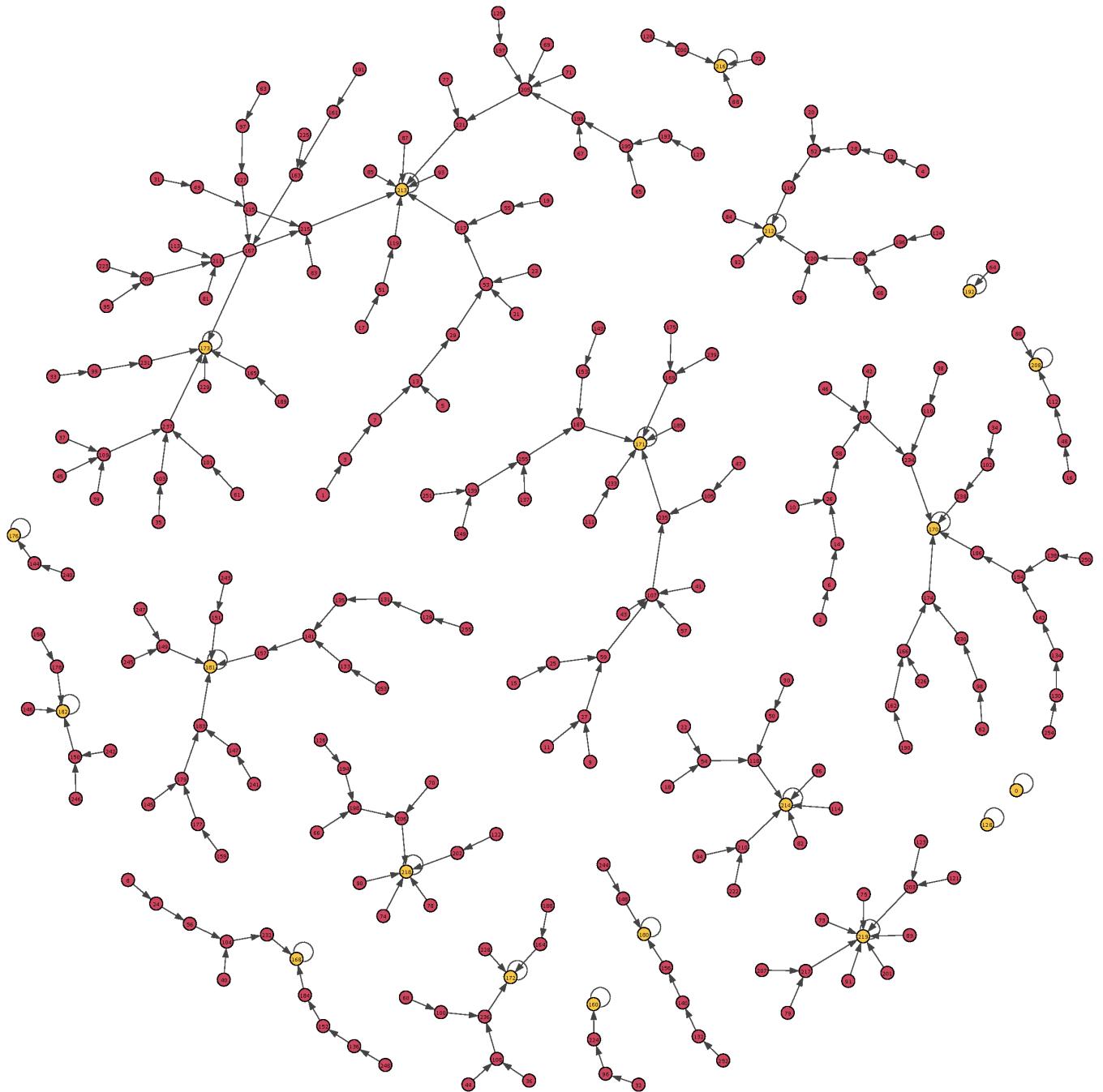


Figure 73: Grafo de los atractores para la regla 78 con $n = 8$

3.1.56 Regla 90

Regla con comportamiento particularmente errático en la construcción de sus campos de atracción. Mostrando para diferentes valores de n , comportamientos particulares de un conjunto de reglas, estas pueden cambiar en la siguiente generación.

Poniendo como ejemplo, en la generación 5 la construcción se realiza de varios árboles con topología extensa. En la siguiente generación su comportamiento es de fabricación de un árbol único con atractor el estado 0, y 2 generaciones posteriores se observa la creación de campos de atracción con ciclos bien definidos para un gran número de campos de atracción.

A pesar de este dinamismo en la regla, se mantiene constante la presencia del atractor 0 entre generaciones, algunas veces como atractor con un árbol de evolución, aveces como atractor único y finalmente en ocasiones como atractor marginal.

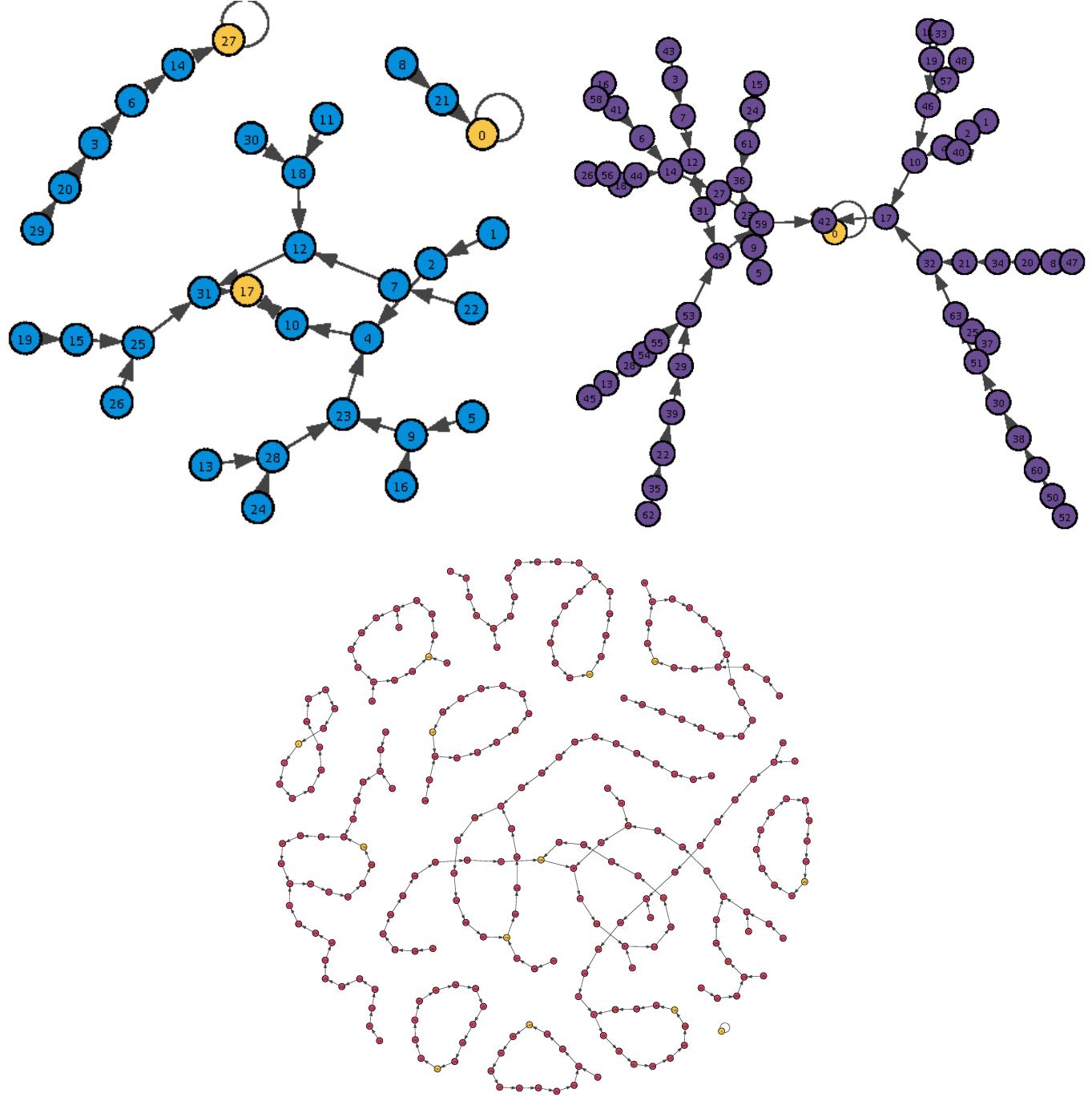


Table 6: Grafo de los atractores para la regla 90 con $n = 5, 6, 8$

3.1.57 Regla 94

La regla 94 describe los campos atractores mediante la construcción de varios árboles de evolución con distribuciones diferentes: algunos de ellos tienen una extensión grande por ramas con nodos consecutivos, otros tantos generan ciclos y finalmente algunos cuentan con nodos con propiedades atractoras y aglomeran ancestros.

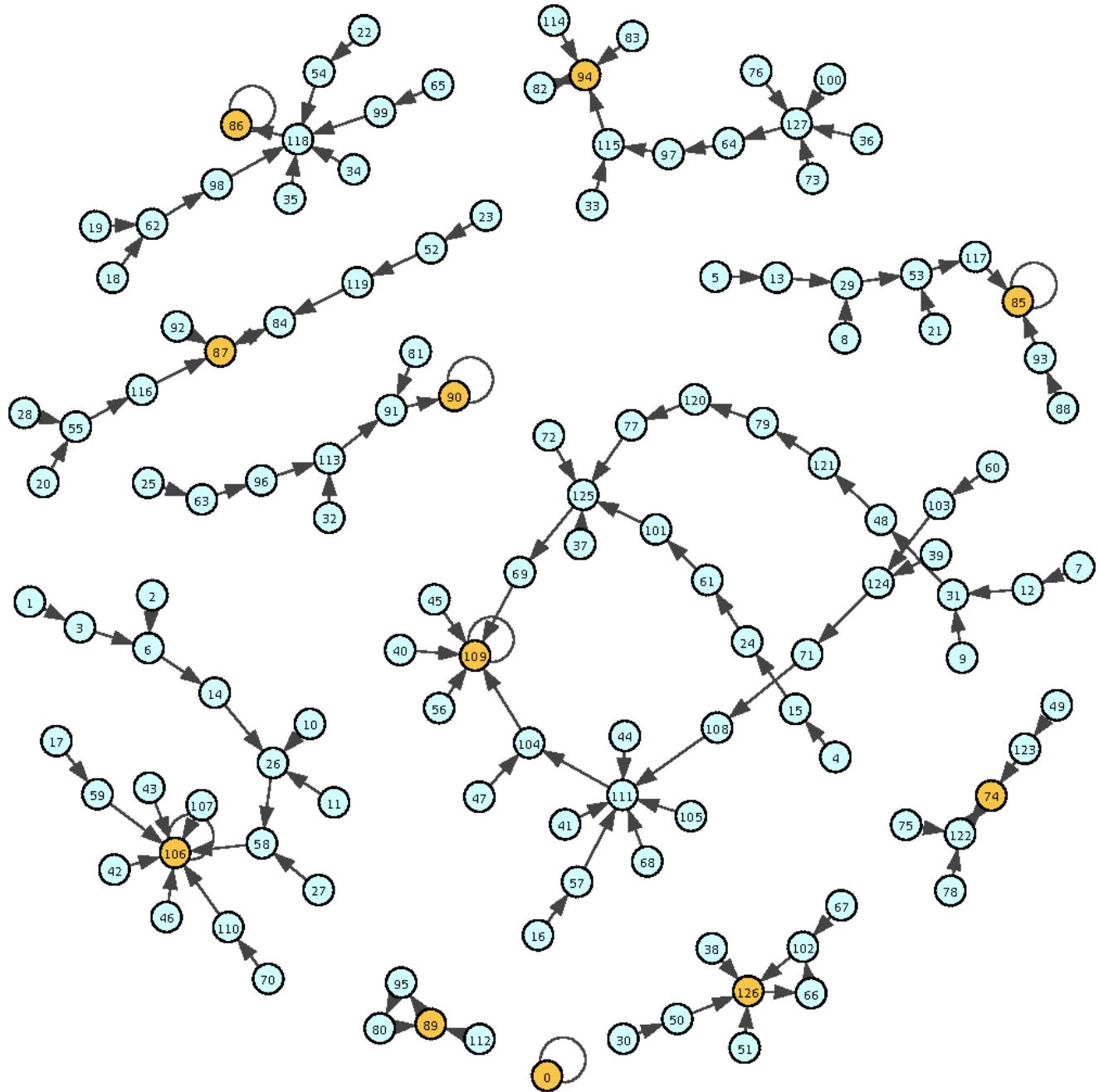


Figure 74: Grafo de los atractores para la regla 94 con $n = 7$

3.1.58 Regla 104

Con el nodo 0 como atractor principal, la regla 104 genera árboles pequeños con ramas de nodos consecutivos sin propiedades atractoras, así como algunos atractores marginales.

De manera general, el tiempo de convergencia es moderado y no se requieren de mucha fases para la estabilización del sistema.

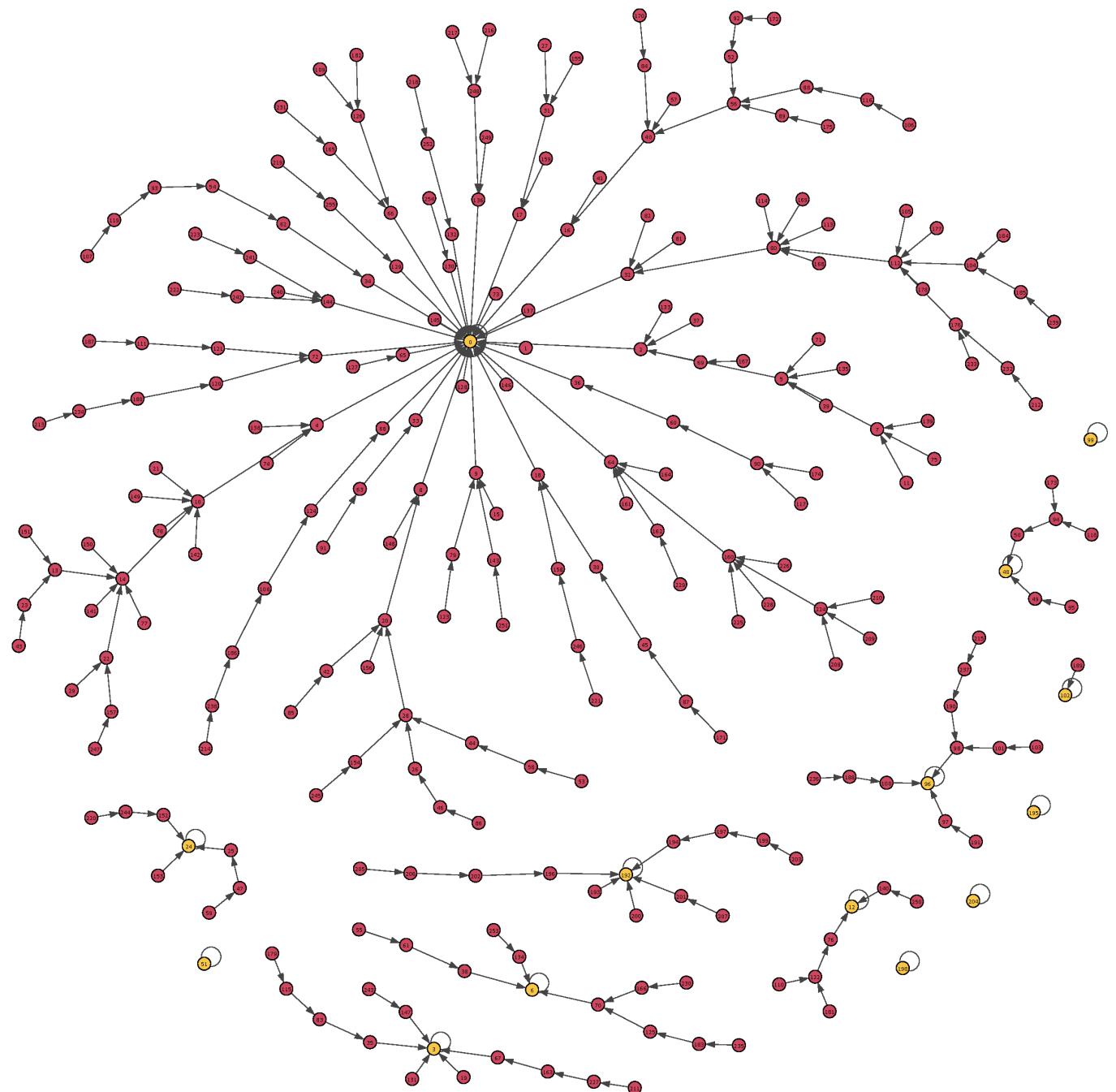


Figure 75: Grafo de los atractores para la regla 104 con $n = 8$

3.1.59 Regla 105

Al igual que sucedía con la regla 90 y $n=8$, esta regla muestra una construcción de sus atractores a través de ciclos de los que les surgen ramas con nodos consecutivos.

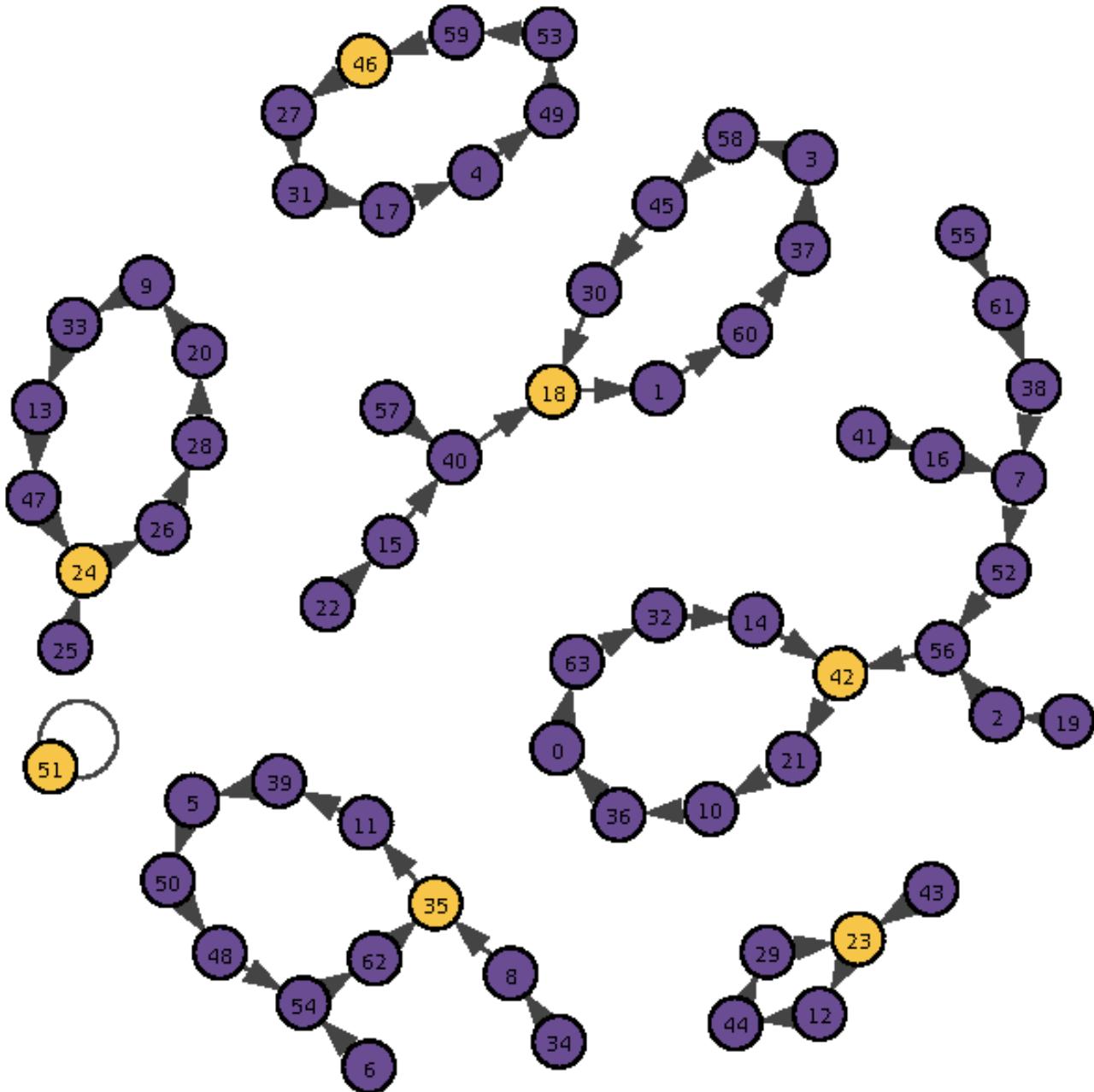


Figure 76: Grafo de los atractores para la regla 105 con $n = 6$

3.1.60 Regla 106

La regla 106 realiza la construcción de un árbol de evolución con el nodo 0 como atractor, más característico este arbol es la derivación de muchas ramas que se construyen para un único atractor. También se puede observar la presencia de un atractor marginal que mantiene su identidad como múltiplo de 3 y 2 al mismo tiempo.

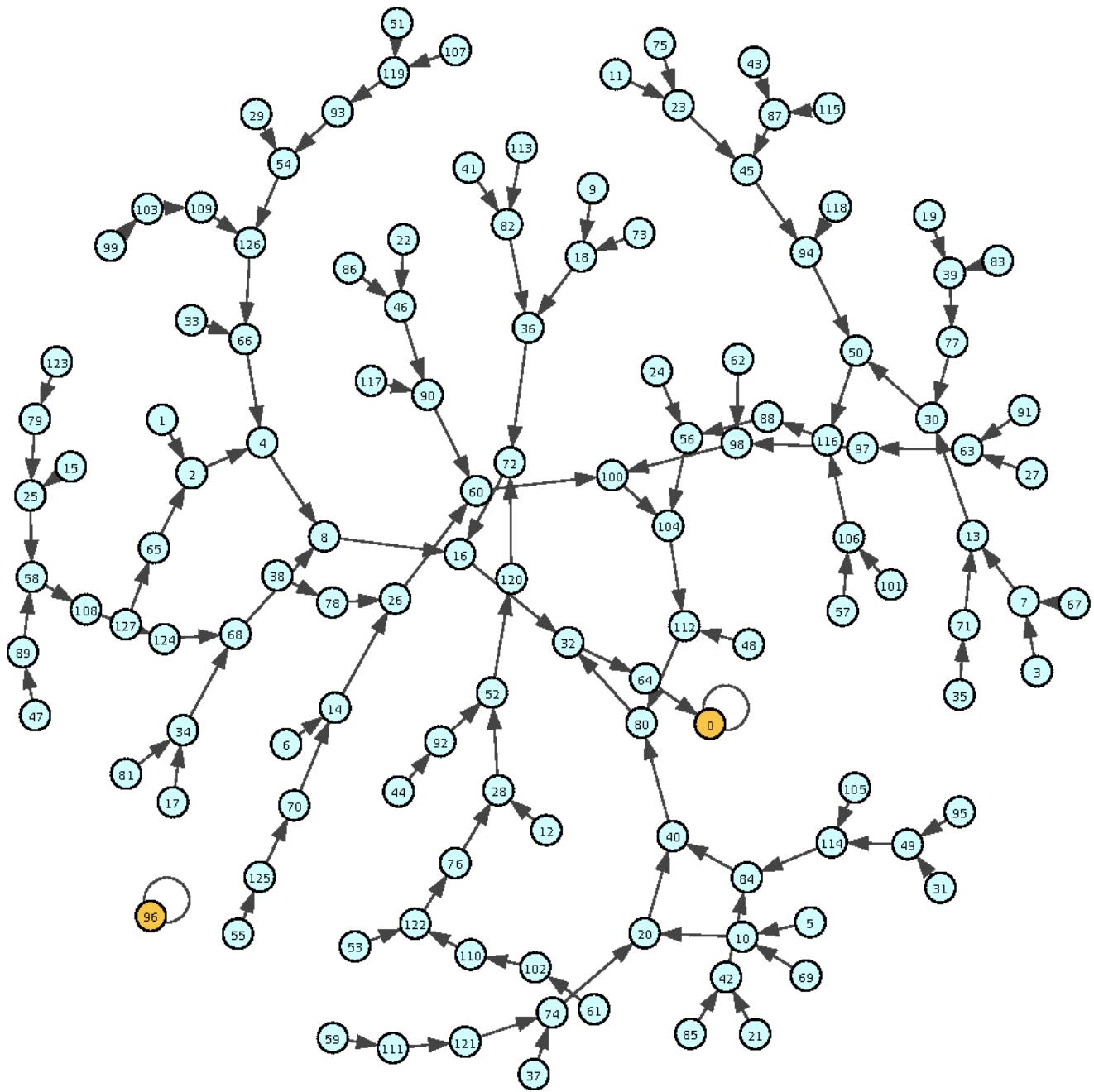


Figure 77: Grafo de los atractores para la regla 106 con $n = 7$

3.1.61 Regla 108

Regla característica por la construcción de sus campos de atracción con una noción geométrica. De forma general las estructuras construidas para esta regla son con una cantidad pequeña de nodos, generando a su vez una gran cantidad de estructuras, destacando además un gran conjunto de atractores marginales presentes en todas las generaciones.

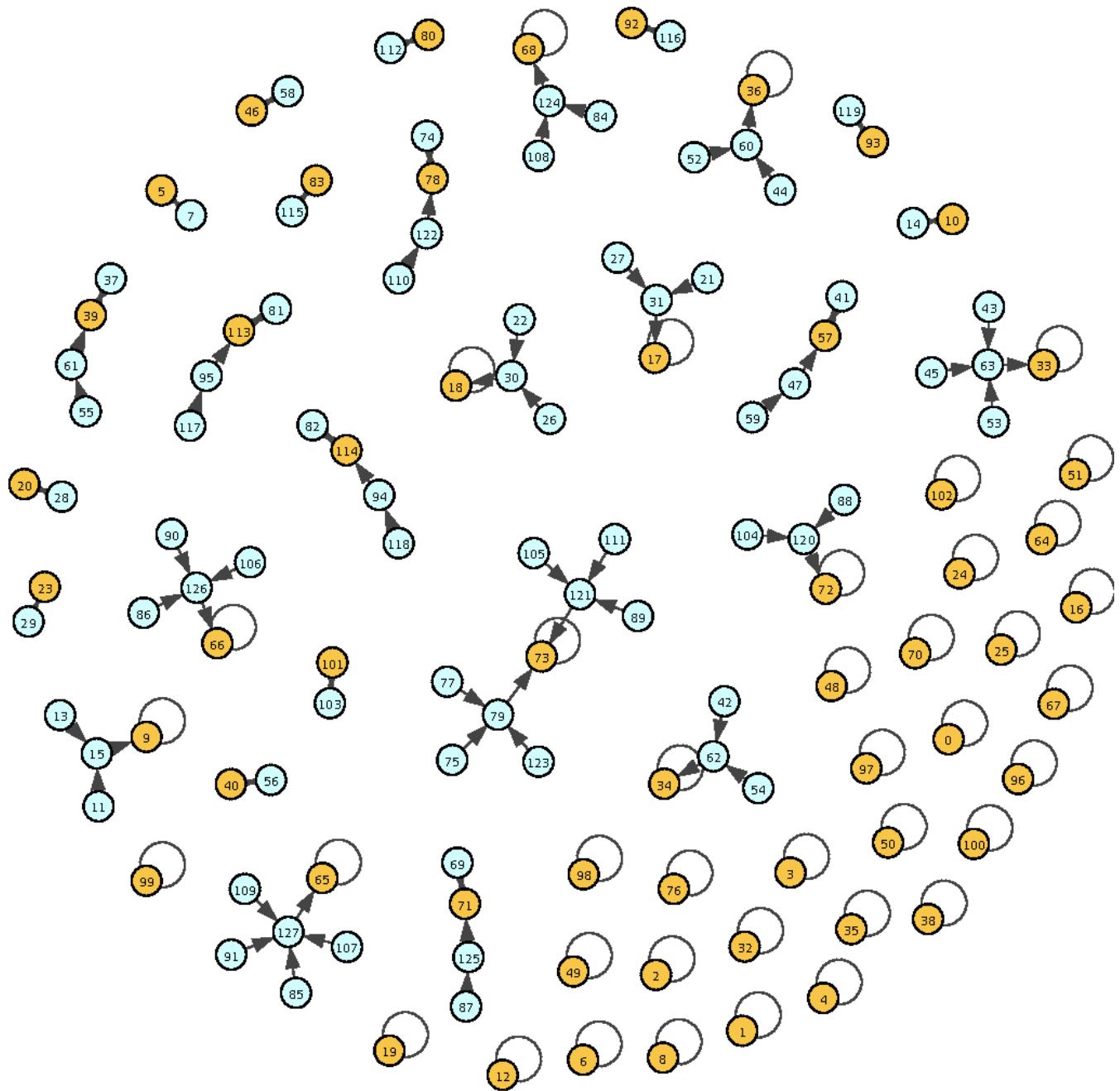


Figure 78: Grafo de los atractores para la regla 108 con $n = 7$

3.1.62 Regla 110

Típica construcción de un moderado número de árboles de evolución con topología extensa. Constantemente se advierte la presencia de un par de atractores marginales con valor 0 y el otro que aumenta como múltiplo de 2.

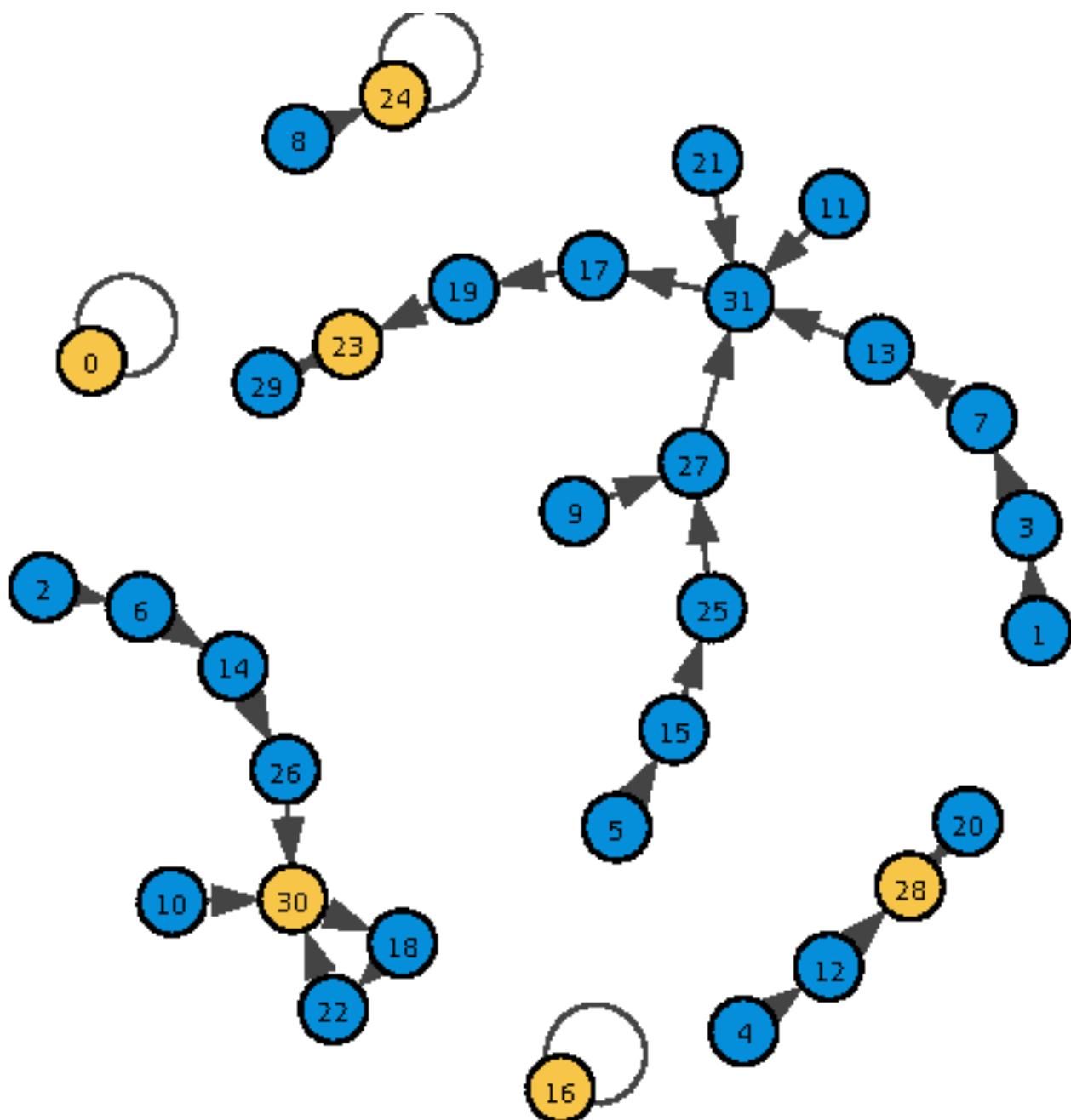


Figure 79: Grafo de los atractores para la regla 110 con $n = 5$

3.1.63 Regla 122

Típica construcción de árboles extensos con tendencia a crecer su número a medida que n aumenta su valor. Con este tipo de topologías de identifica una velocidad de convergencia lenta.

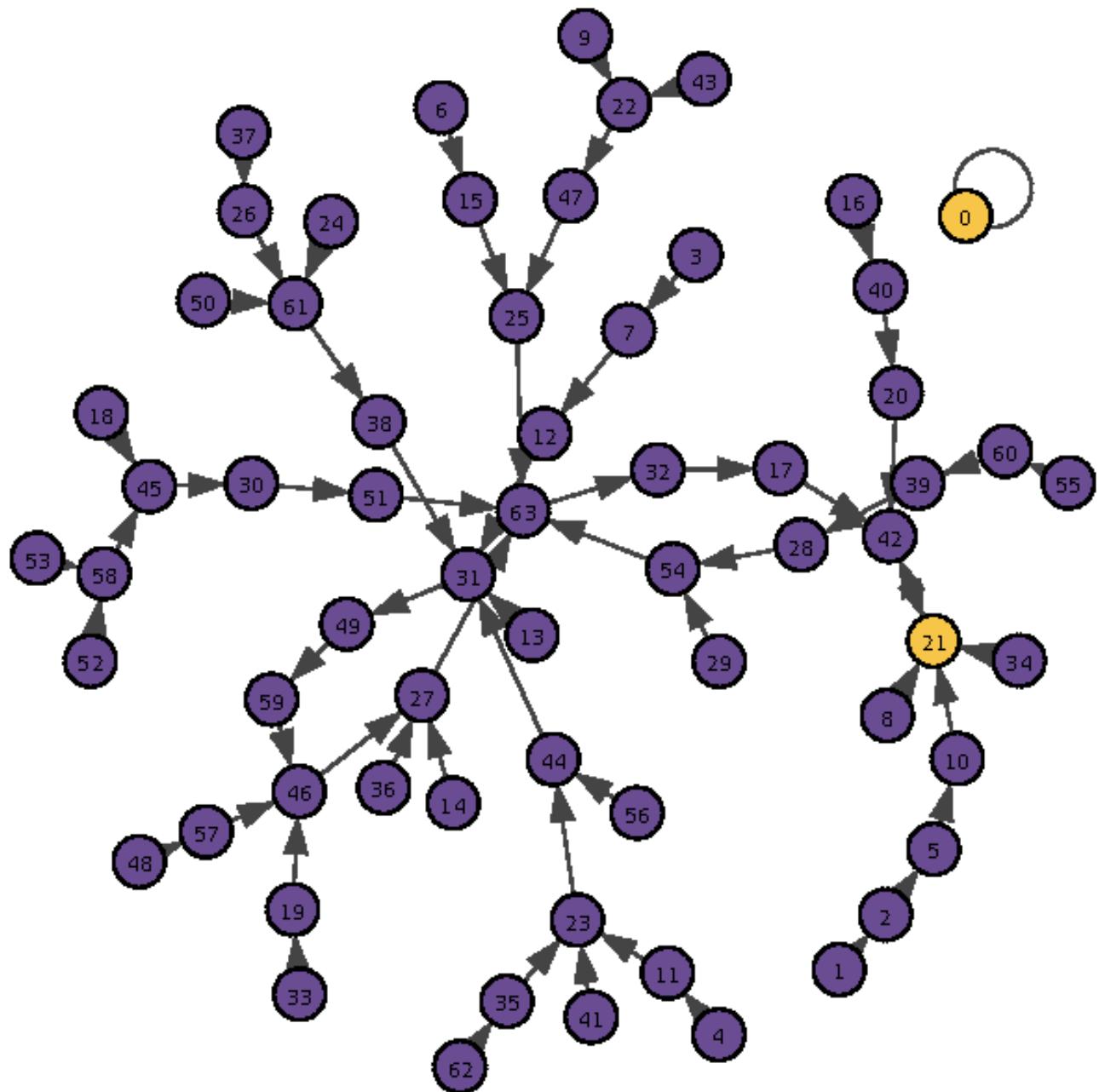


Figure 80: Grafo de los atractores para la regla 122 con $n = 6$

3.1.64 Regla 126

Interesante comportamiento de la regla donde se construye un árbol principal con un nodo con una fuerte propiedad de atracción, pudiendo ser o no, este nodo un atractor.

Fuera de este árbol principal, se crean algunos árboles pequeños y otros campos de atracción con nodos que también comparte la propiedad atractora aglomeando nodos ancestros.

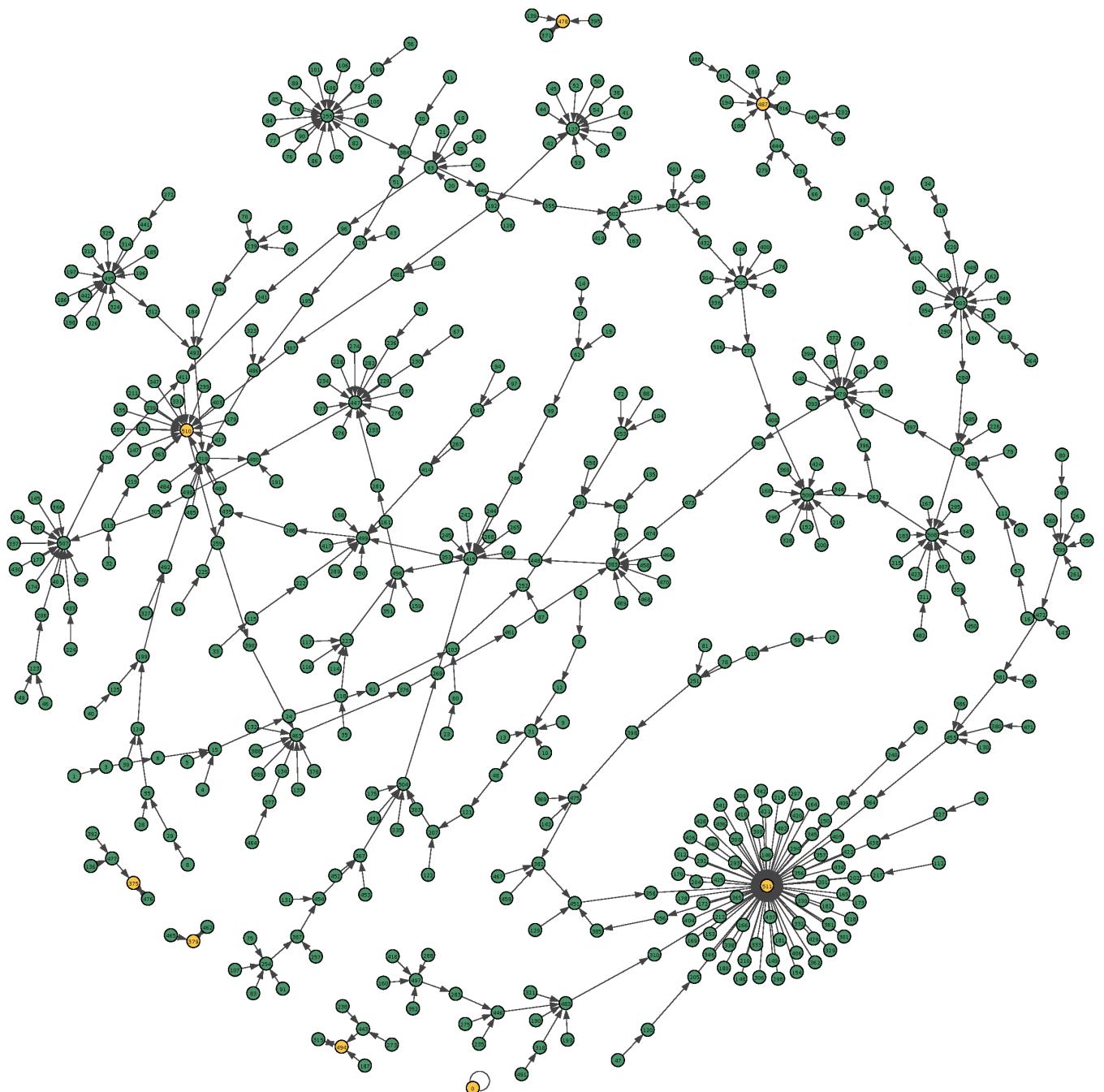


Figure 81: Grafo de los atractores para la regla 126 con $n = 9$

3.1.65 Regla 128

Regla con un comportamiento parecido a las 8, 40, pero más similar a la 32, su evolución construye un árbol de evoluciones con el nodo 0 como atractor principal. Contando con una fuerte propiedad de atracción el nodo 0 y aglomerando gran cantidad de ancestros directos como nodos raíz, también se identifican ramas que cuentan con nodos extremos que igual cuentan con la propiedad de atracción pero en menor medida.

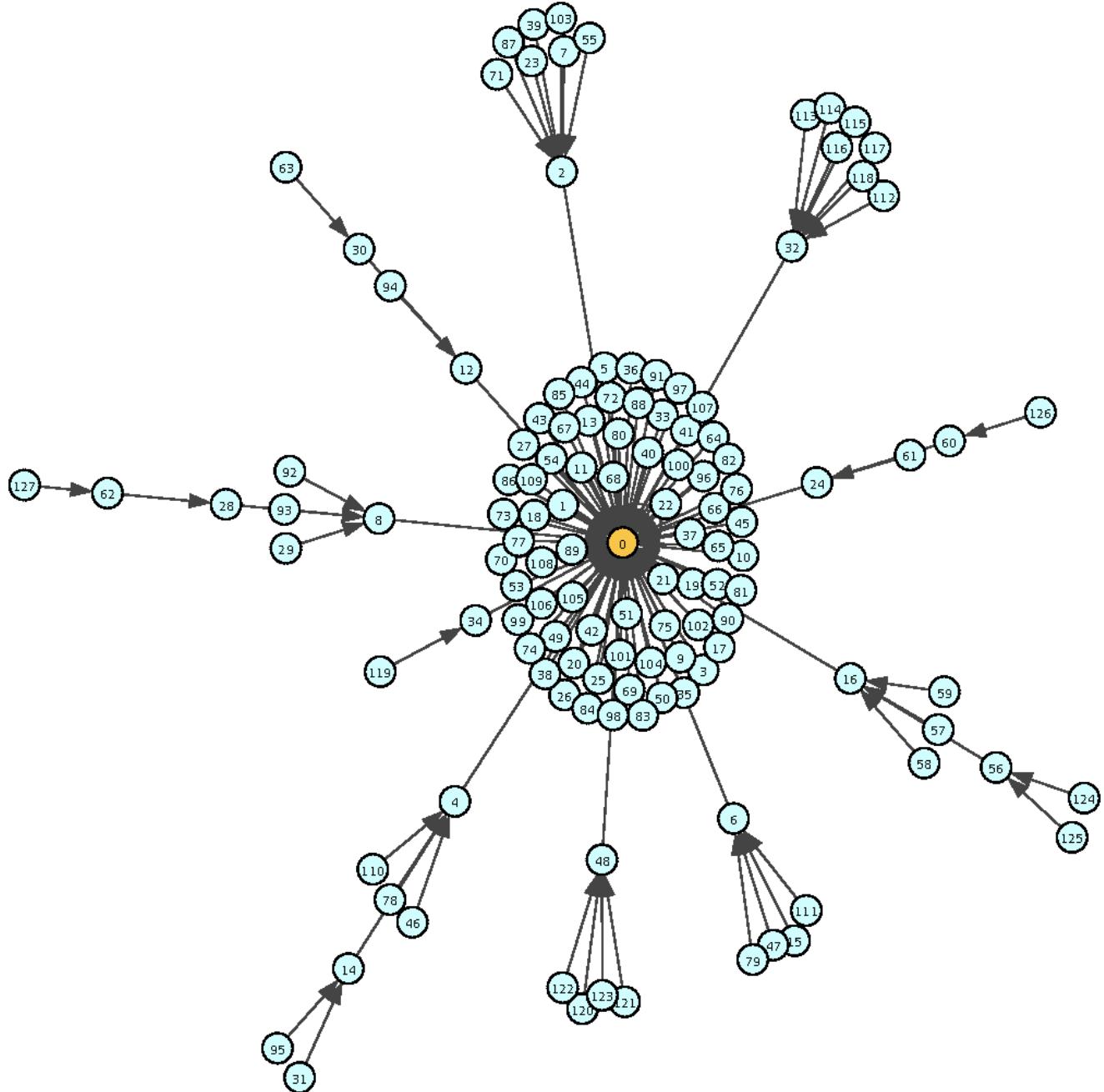


Figure 82: Grafo del atractor para la regla 128 con $n = 7$

3.1.66 Regla 130

Con comportamiento similar a otras reglas, se genera un árbol único con el nodo 0 como atractor principal, a pesar de esto, los nodos en la rama principal cuentan con la propiedad atractora y aglomeran nodos ancestros.

El tiempo de convergencia con la topología de este árbol es moderada pues si se tienen ramas con nodos consecutivos, pero no son cadenas exageradamente extensas.

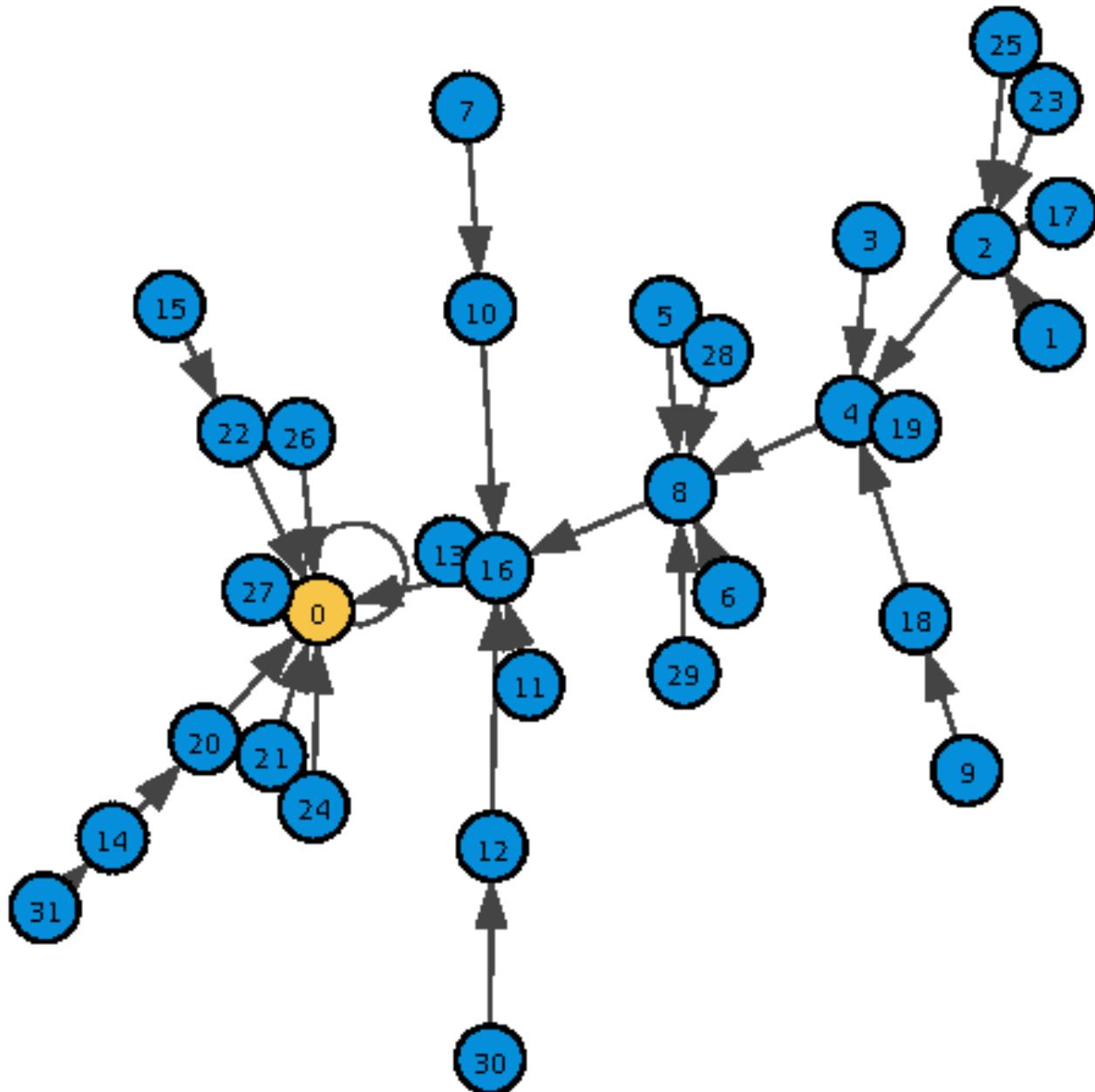


Figure 83: Grafo del atractor para la regla 130 con $n = 5$

3.1.67 Regla 132

Regla constructora de una gran número de campos de atracción con un conjunto limitado de nodos por cada uno, a excepción del árbol principal con el nodo 0 como atractor, la cual con una importante capacidad de atracción, aglomera un número de ancestros directamente de las ramas que se unen a este.

También se observa una presencia constante de atractores marginales, y que en términos generales va a propiciar a que el número de evoluciones requeridas para la convergencia sea moderadamente pocas.

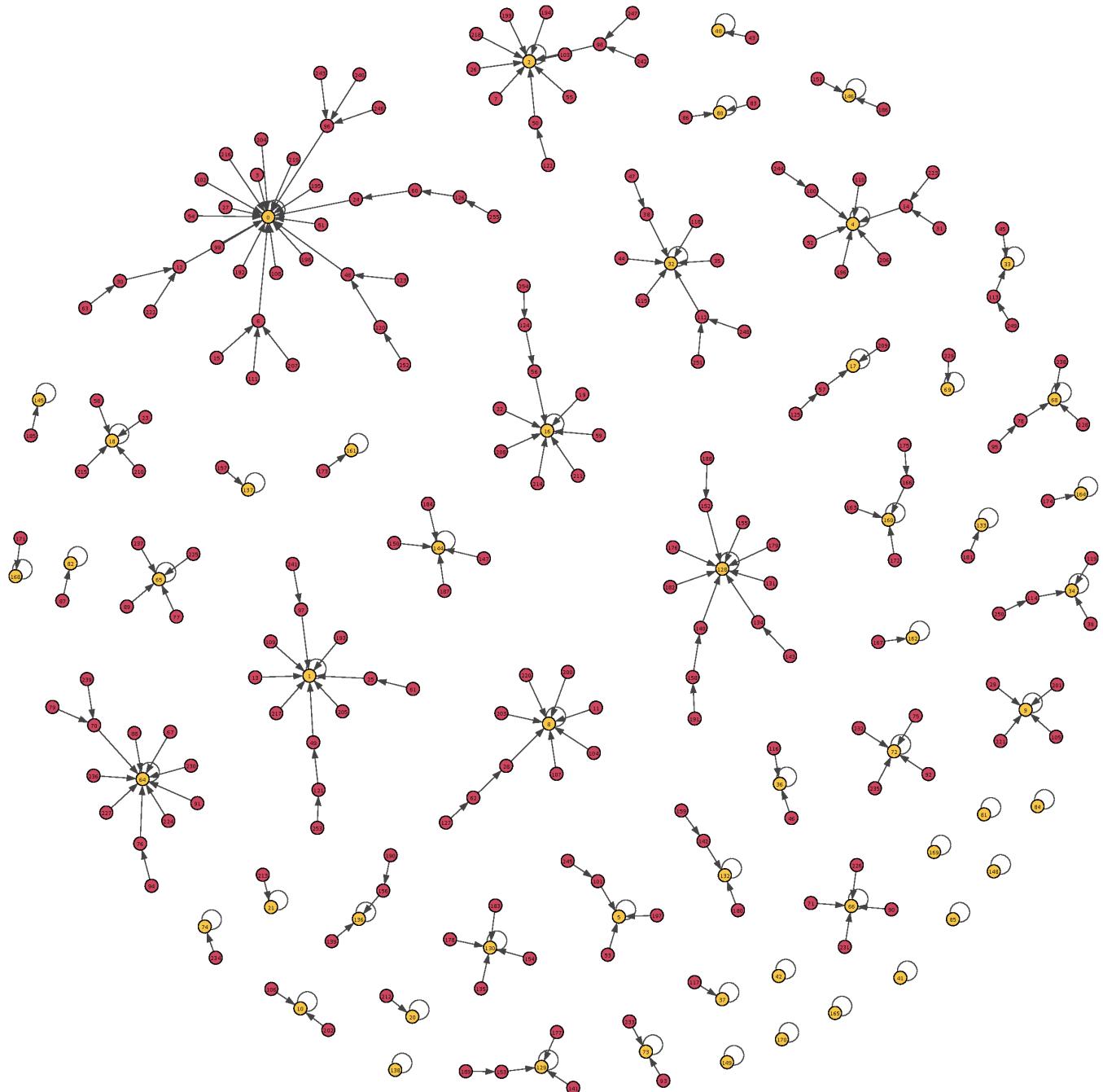


Figure 84: Grafo de los atractores para la regla 132 con $n = 8$

3.1.68 Regla 134

La construcción de los campos de atracción para la regla 134 es típica, con una tendencia a crecer el número de árboles que se crean, estos cuentan con una topología extensa con ramas de nodos consecutivos. Destaca de entre los árboles el principal que tiene el nodo 0 como atractor y que acapara la mayor cantidad de nodos en su construcción.

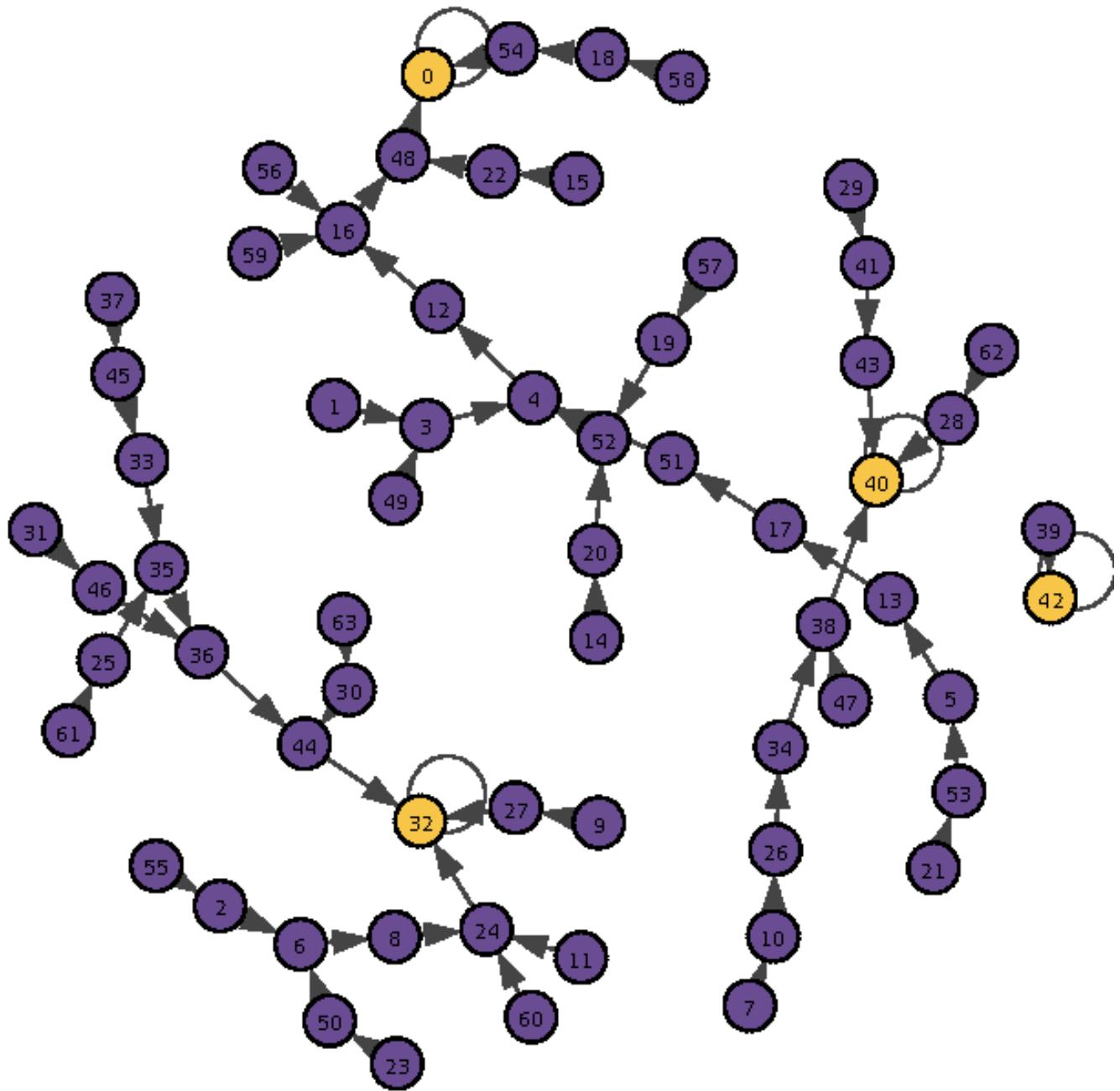


Figure 85: Grafo del atractor para la regla 134 con $n = 6$

3.1.69 Regla 136

El árbol generado en la regla 136 es similar a otros generados por reglas anteriormente mencionadas. El atractor 0 funge como nodo principal con gran atracción conjunto un numero importante de ancestros, pero a pesar de esto se cuentan con ramas largas de nodos consecutivos.

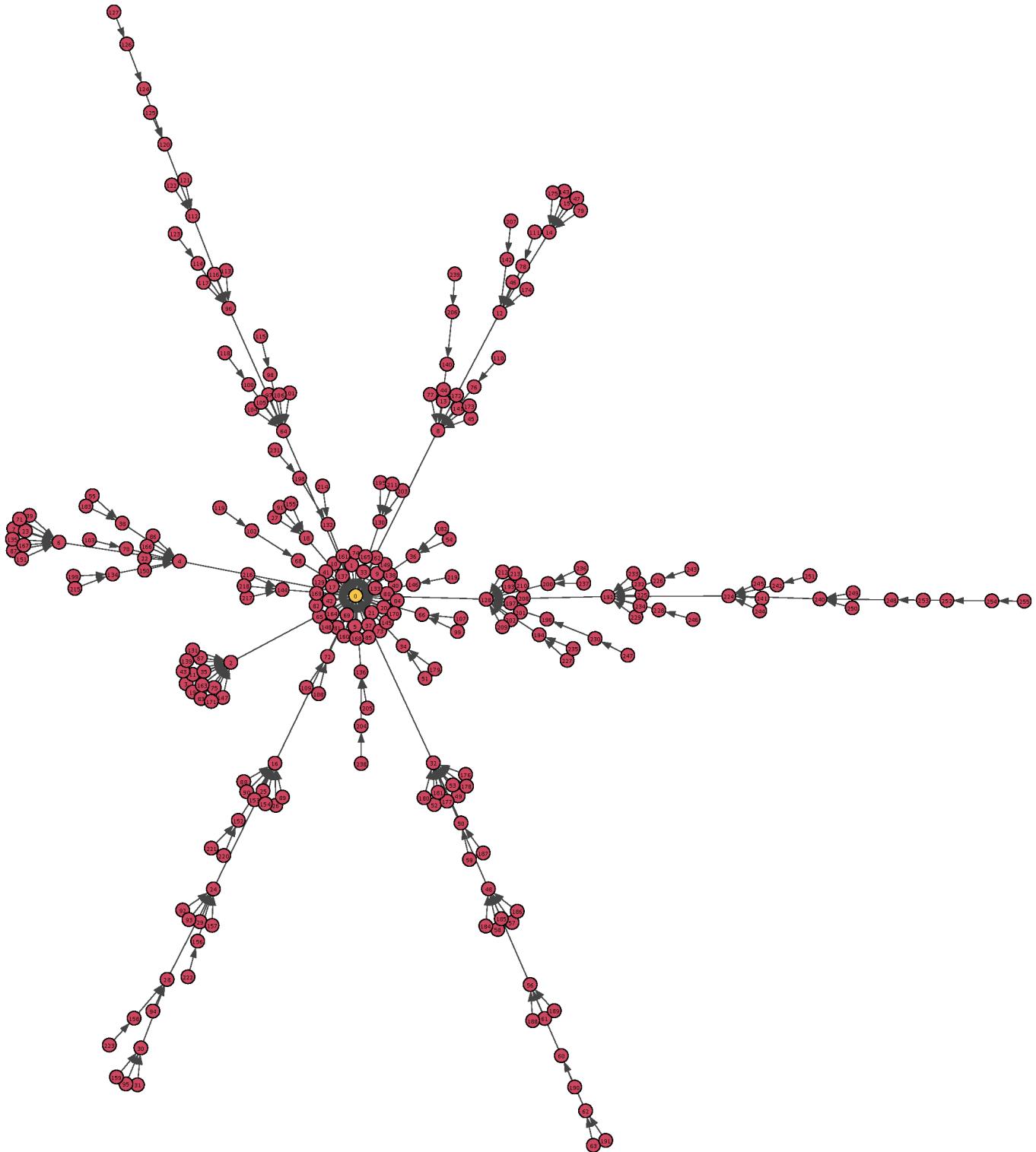


Figure 86: Grafo del atractor para la regla 136 con $n = 8$

3.1.70 Regla 138

Grafo único con nodos en ramas del árbol con propiedades atractoras, siendo más marcada esta propiedad en los nodos extremos, aunque también en el nodo atractor 0.

En términos generales la velocidad de convergencia es moderada a causa de las ramas, que aunque existen, no son tan extensas con cadenas de nodos consecutivos.

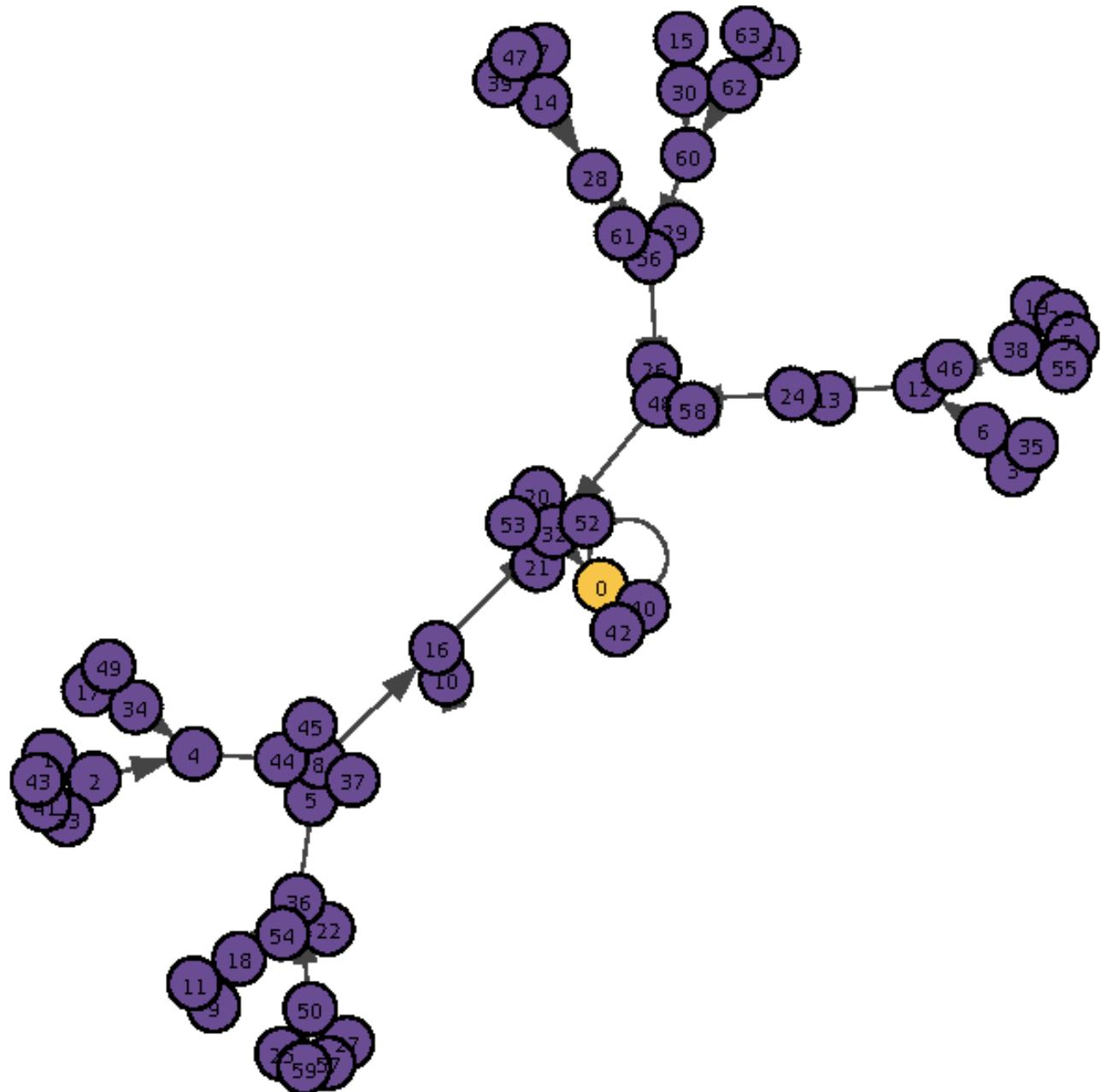


Figure 87: Grafo del atractor para la regla 138 con $n = 6$

3.1.71 Regla 140

La regla 140 describe campos de atracción cortos, con cadenas secuenciales de nodos en configuraciones no muy grandes aunque bastas en número. Así mismo se localizan la presencia de atractores marginales.

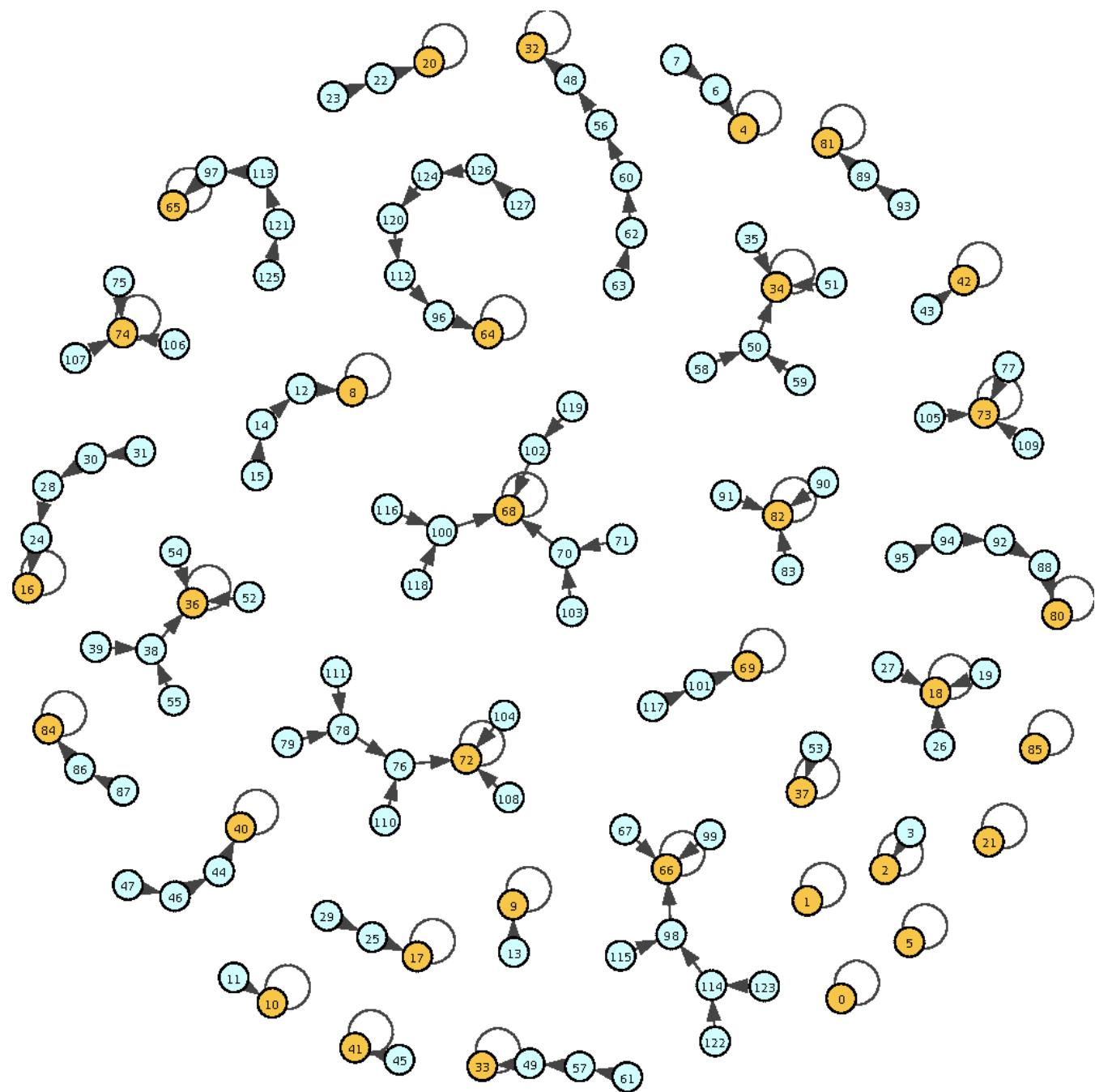


Figure 88: Grafo de los atractores para la regla 140 con $n = 7$

3.1.72 Regla 142

Configuraciones típicas con construcción de árboles con una topología extensa y grandes cadenas de nodos consecutivos, destacando un árbol con atractor principal por el número de nodos que lo conforman. También se advierte la presencia de al menos 1 atractor marginal en cada generación.

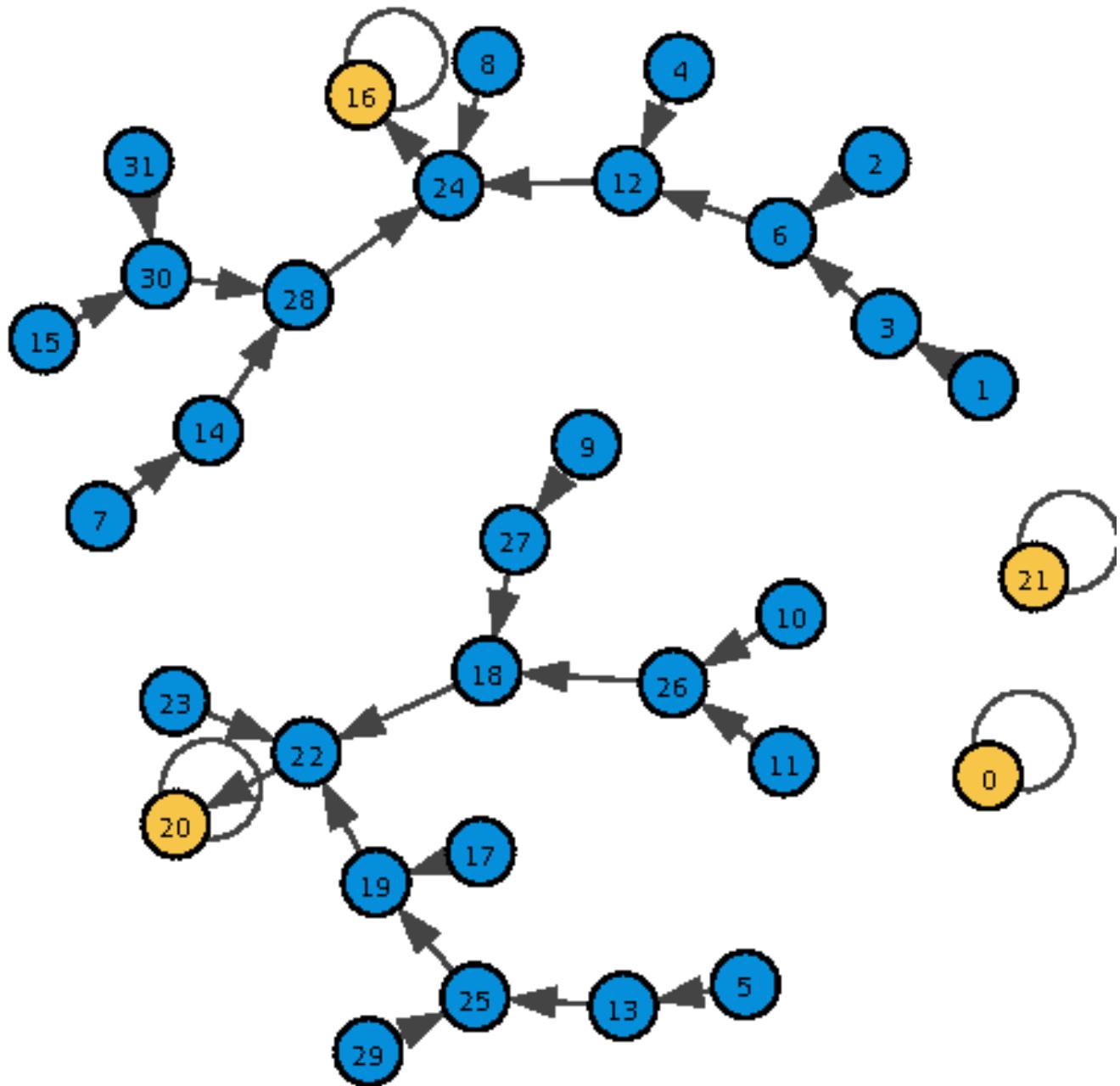


Figure 89: Grafo de los atractores para la regla 142 con $n = 5$

3.1.73 Regla 146

Se observa una dinámica variante sin un patrón bien definido en la construcción de los campos de atracción para diferentes valores de n en la regla 146. En algunas de las generaciones se observan árboles con cadenas de nodos consecutivos en cantidades mayores a 1, en otras generaciones árboles únicos con el nodo 0 como atractor principal y otros tanto árboles con nodos con propiedades de atracción.

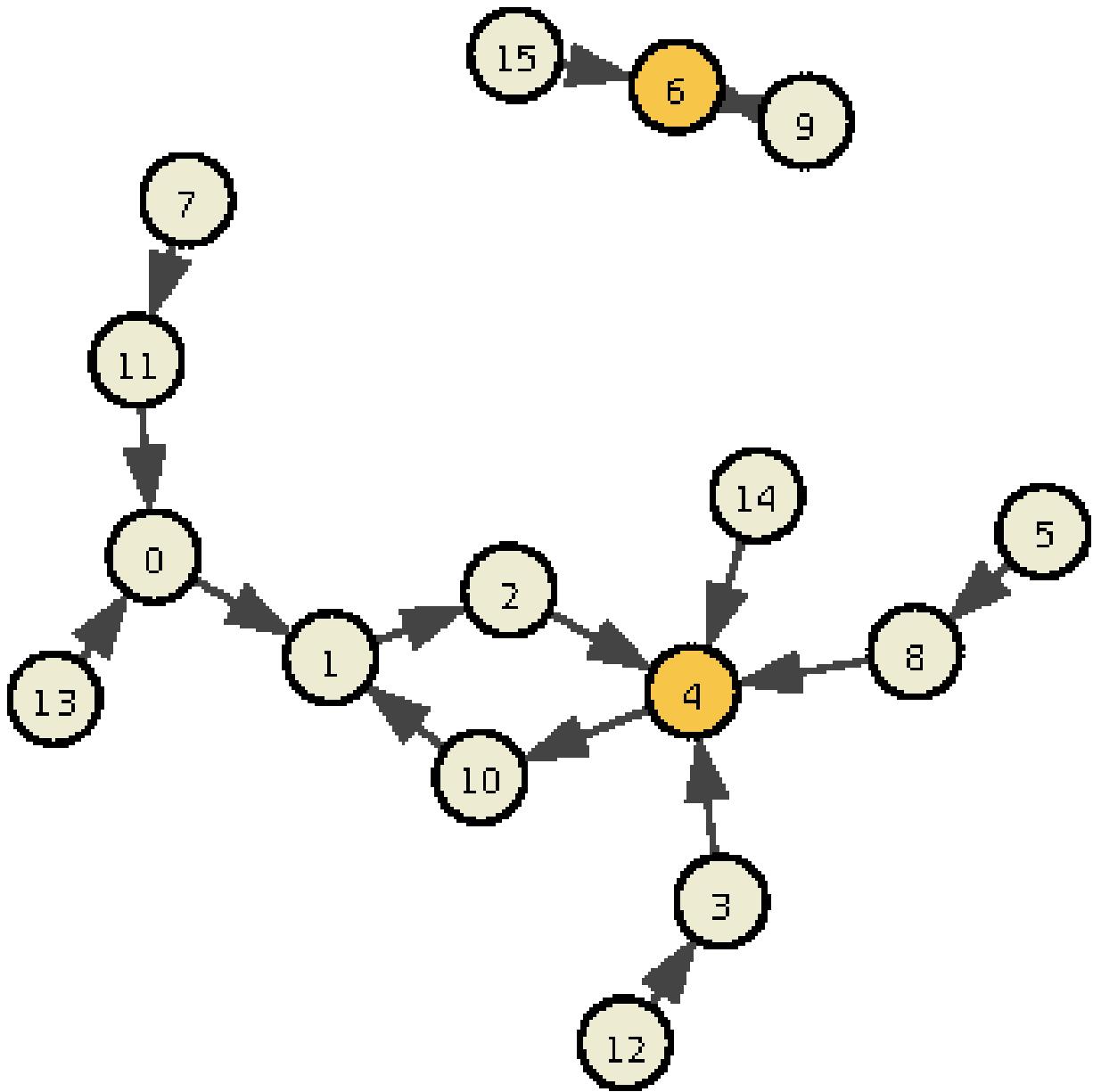


Figure 90: Grafo de los atractores para la regla 146 con $n = 4$

3.1.74 Regla 150

Regla fabricante de árboles de evolución con topologías extensas sin propiedades aglomerantes en nodos específicos. Cuenta con una tendencia creciente en el número de árboles que se generar a medida que n aumenta también. Se advierte la presencia constante del atractor 0 como marginal.

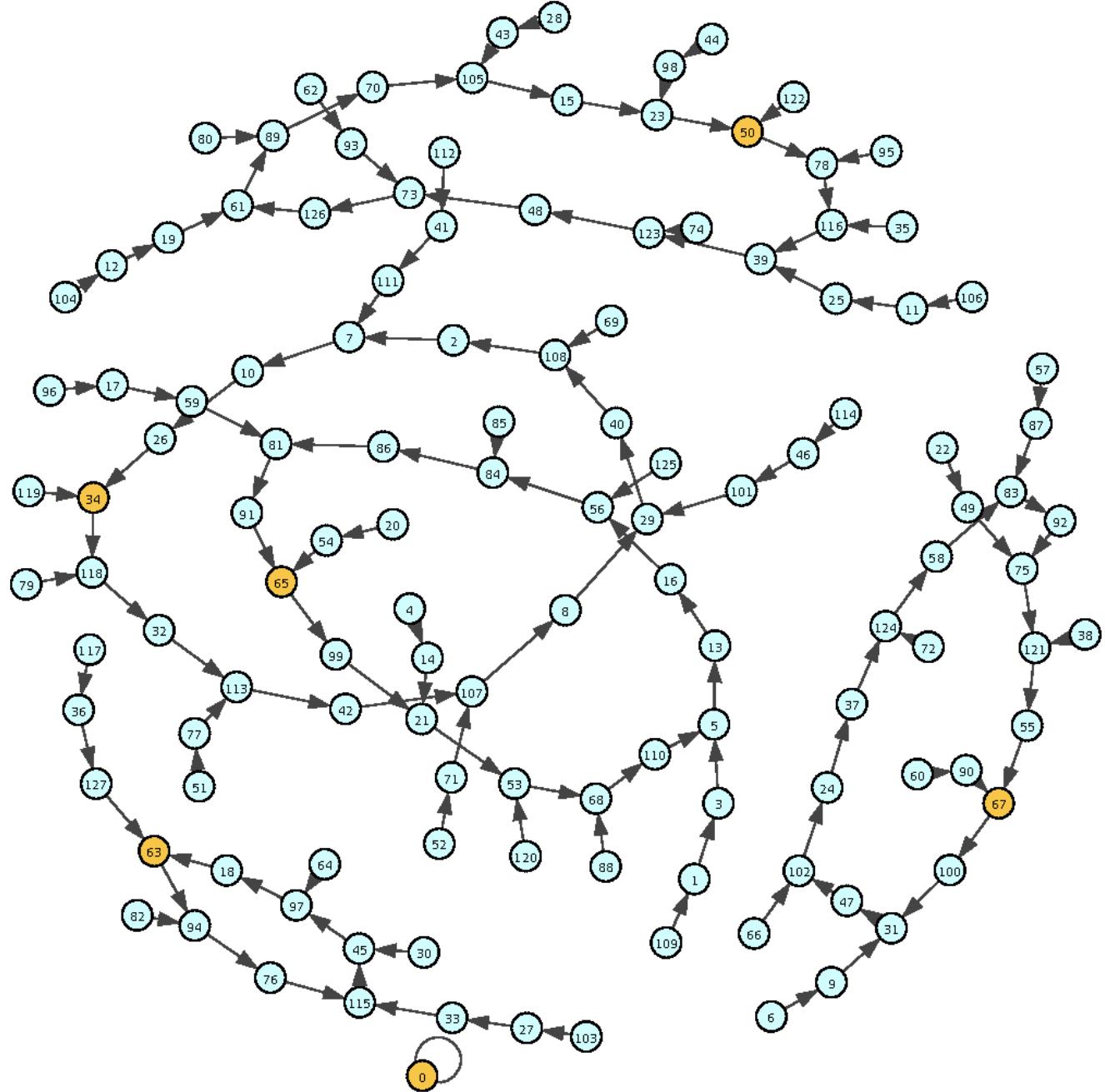


Figure 91: Grafo de los atractores para la regla 150 con $n = 7$

3.1.75 Regla 152

La regla 152 construye un único árbol con el nodo 0 como atractor principal y se caracteriza por constar con una rama principal de larga extensión y de la cual se le unen demás ramas. De forma general su velocidad de estabilización es moderada por el número de cadenas consecutivas de nodos.

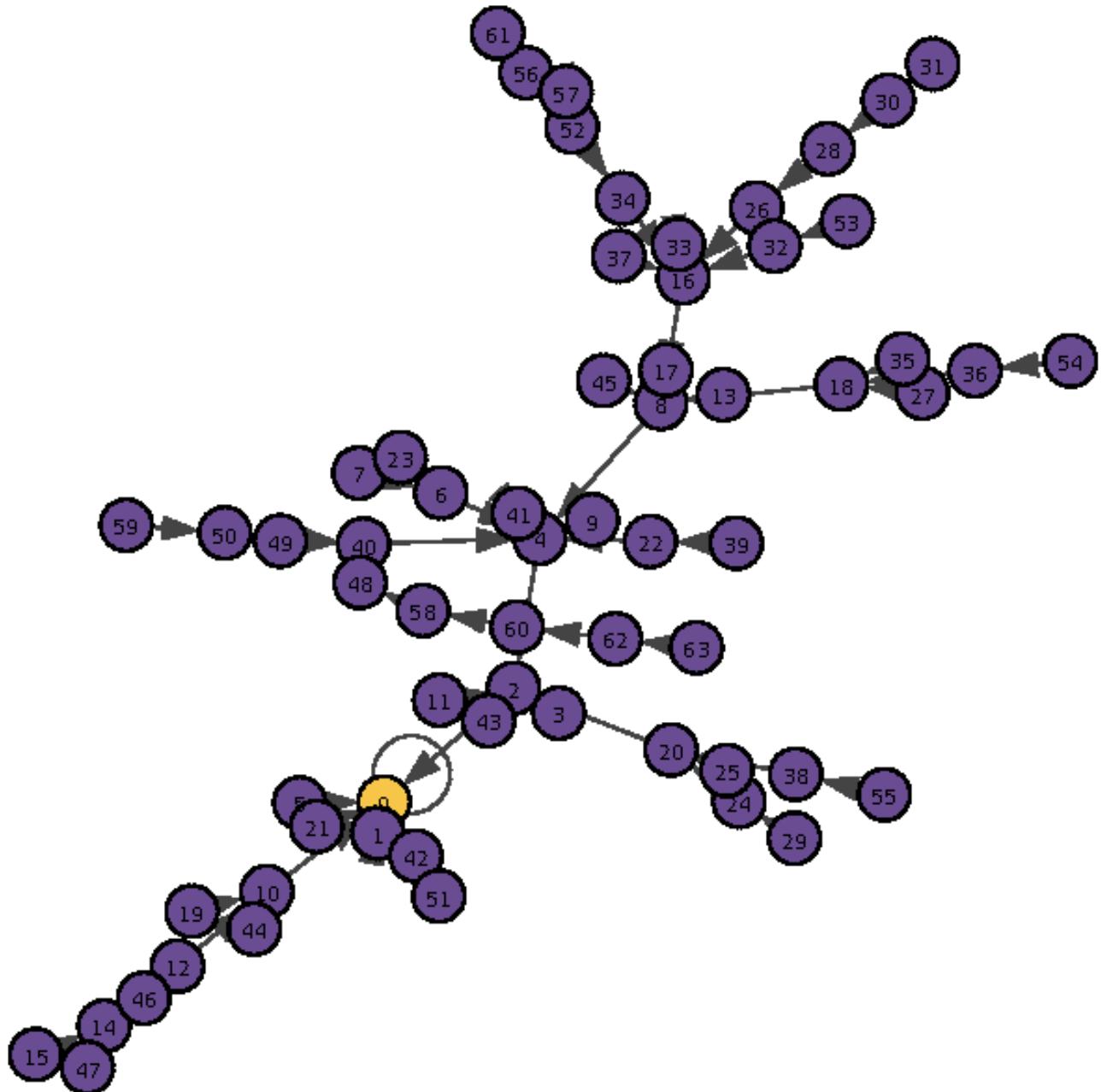


Figure 92: Grafo de los atractores para la regla 152 con $n = 6$

3.1.76 Regla 154

Con un comportamiento igualmente errático sin patrón identificable a la regla 146, para diferentes valores de n los árboles construidos algunas veces son extensos y en número mayor a 1, mientras que en otros casos son únicos y con el nodo 0 como atractor principal.

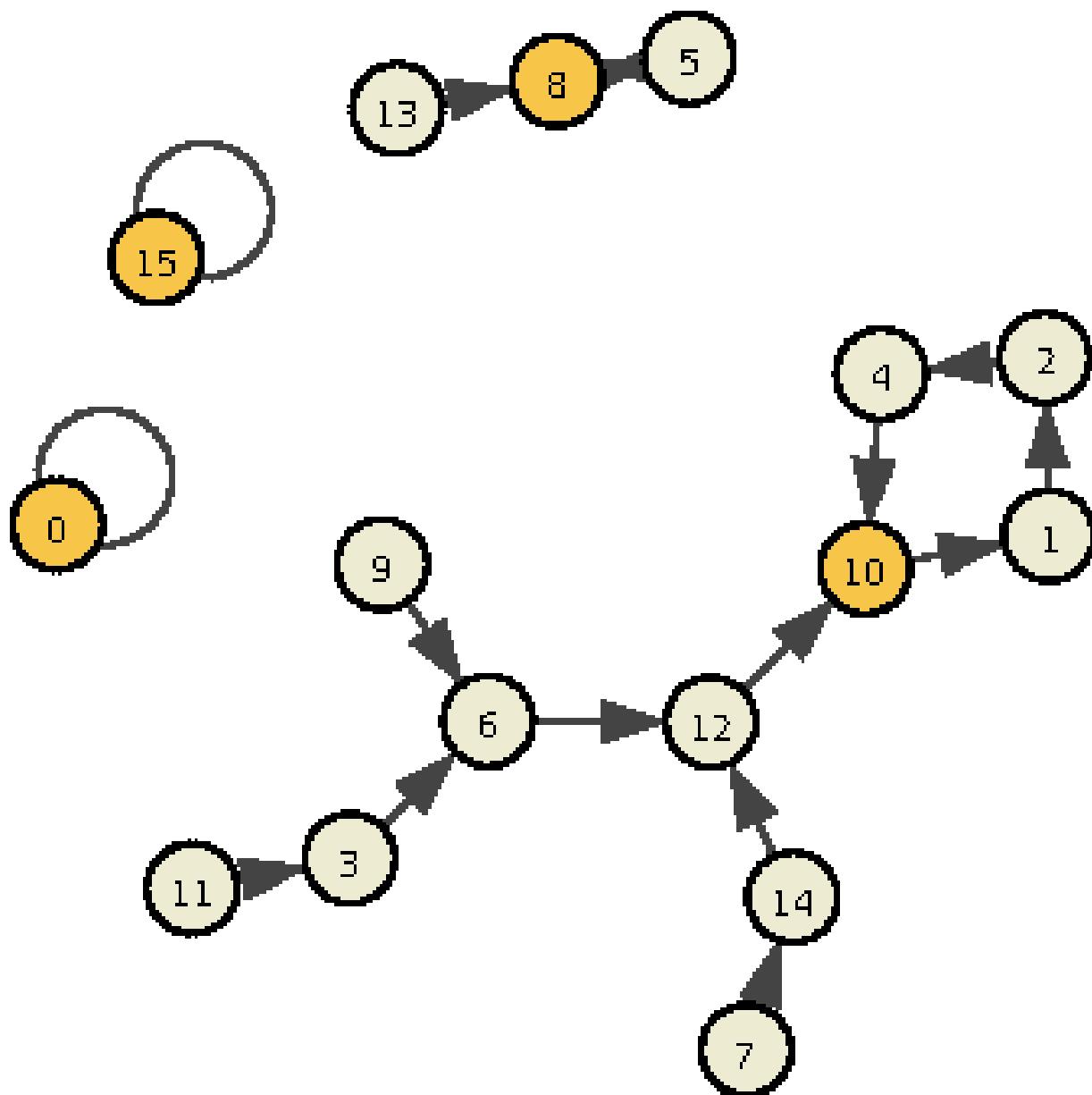


Figure 93: Grafo de los atractores para la regla 154 con $n = 4$

3.1.77 Regla 156

Esta regla describe sus campos de atracción como árboles con ramas de nodos consecutivos sin propiedades atractoras en algún nodo presente. También se advierte la presencia de atractores marginales en cada evolución.

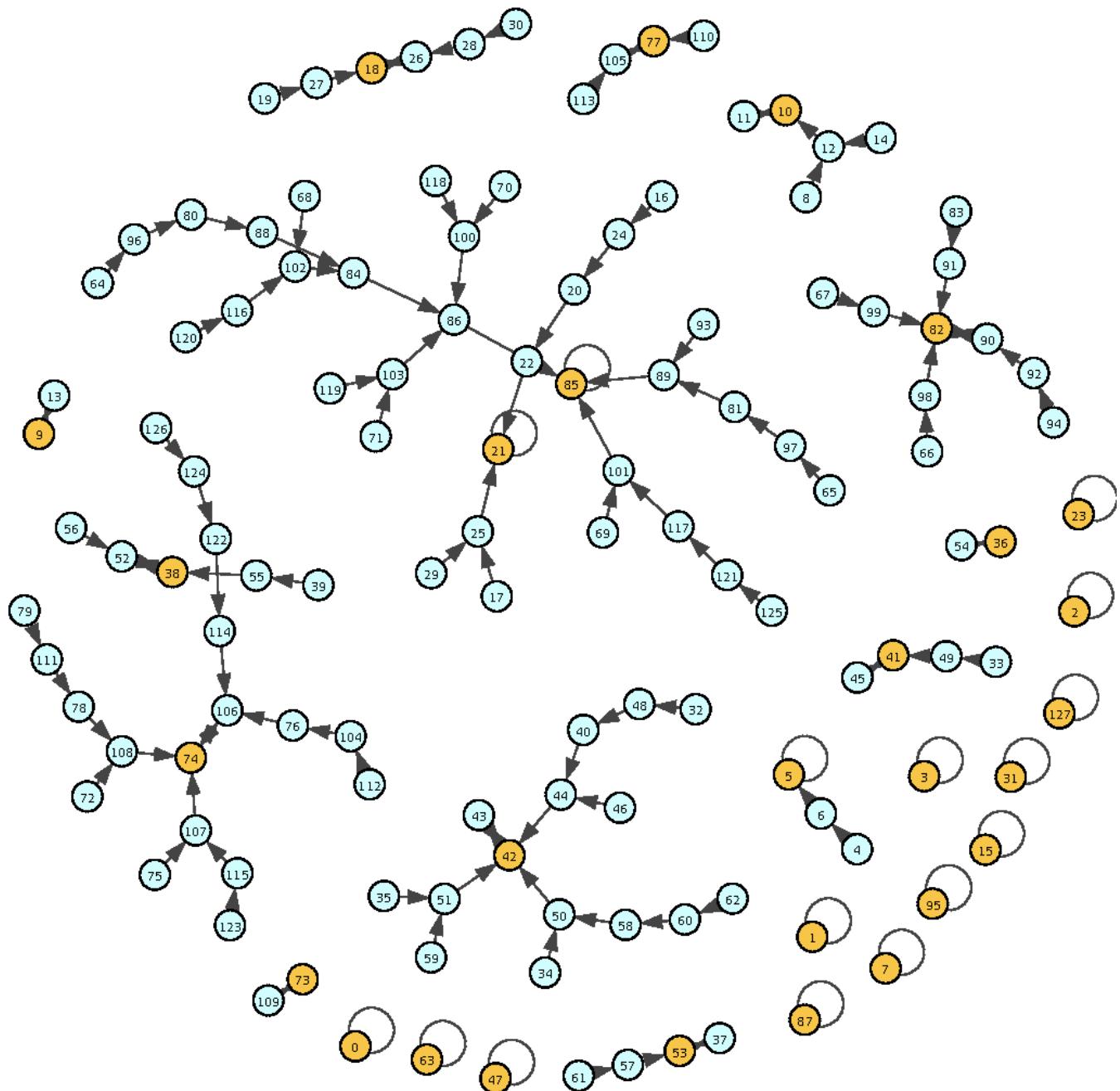


Figure 94: Grafo de los atractores para la regla 156 con $n = 7$

3.1.78 Regla 160

Similar a la regla 128 y 32, se trata de un árbol único con el nodo 0 como atractor principal del que derivan ramas con nodos que también poseen la cualidad de atracción.

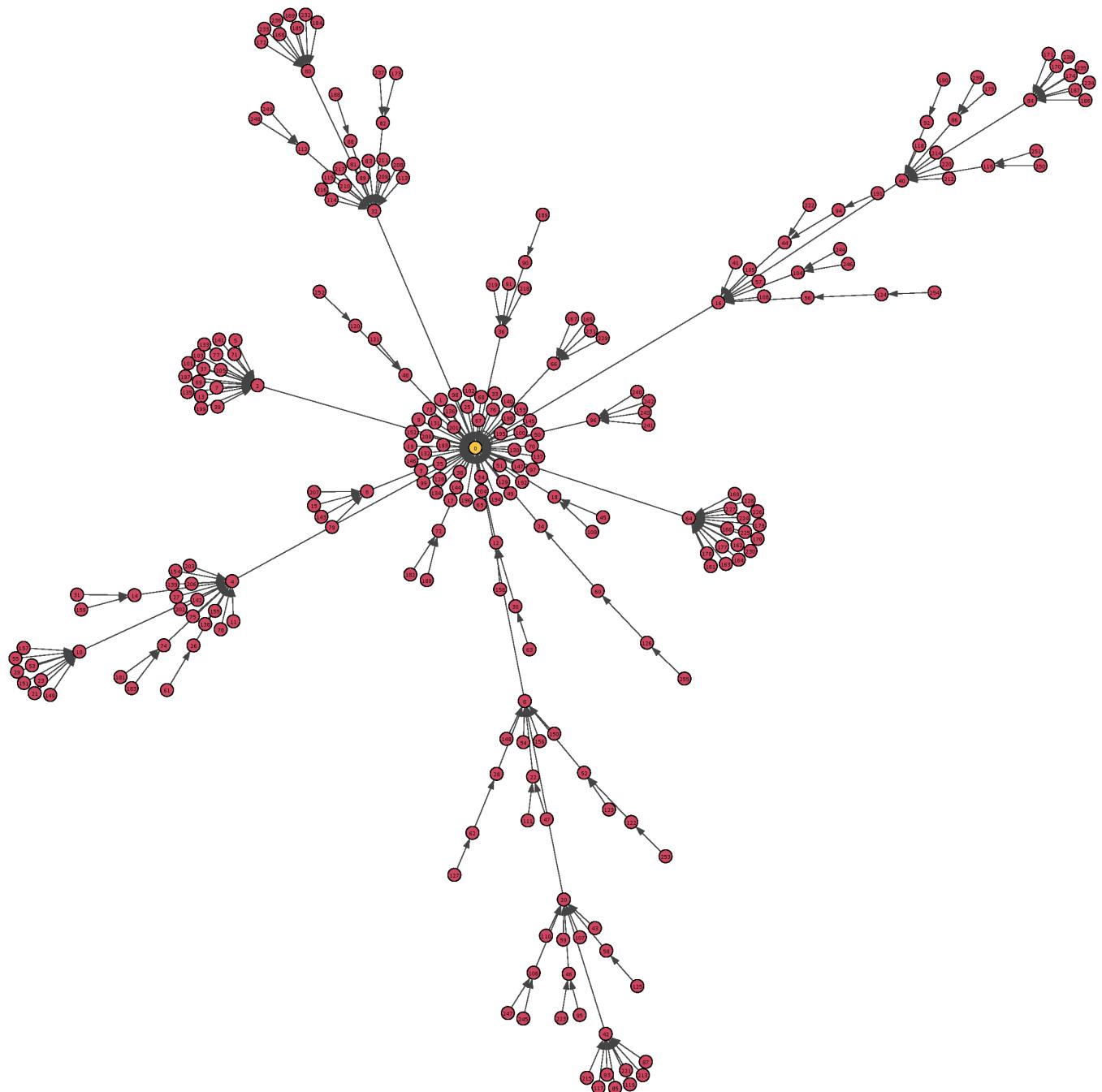


Figure 95: Grafo del atractor para la regla 160 con $n = 8$

3.1.79 Regla 162

Árbol único con el nodo 0 como atractor principal. El propio atractor 0 no muestra cualidades de atracción aunque los nodos en los extremos de las ramas si, aglomerando hojas del edén como nodos ancestros.

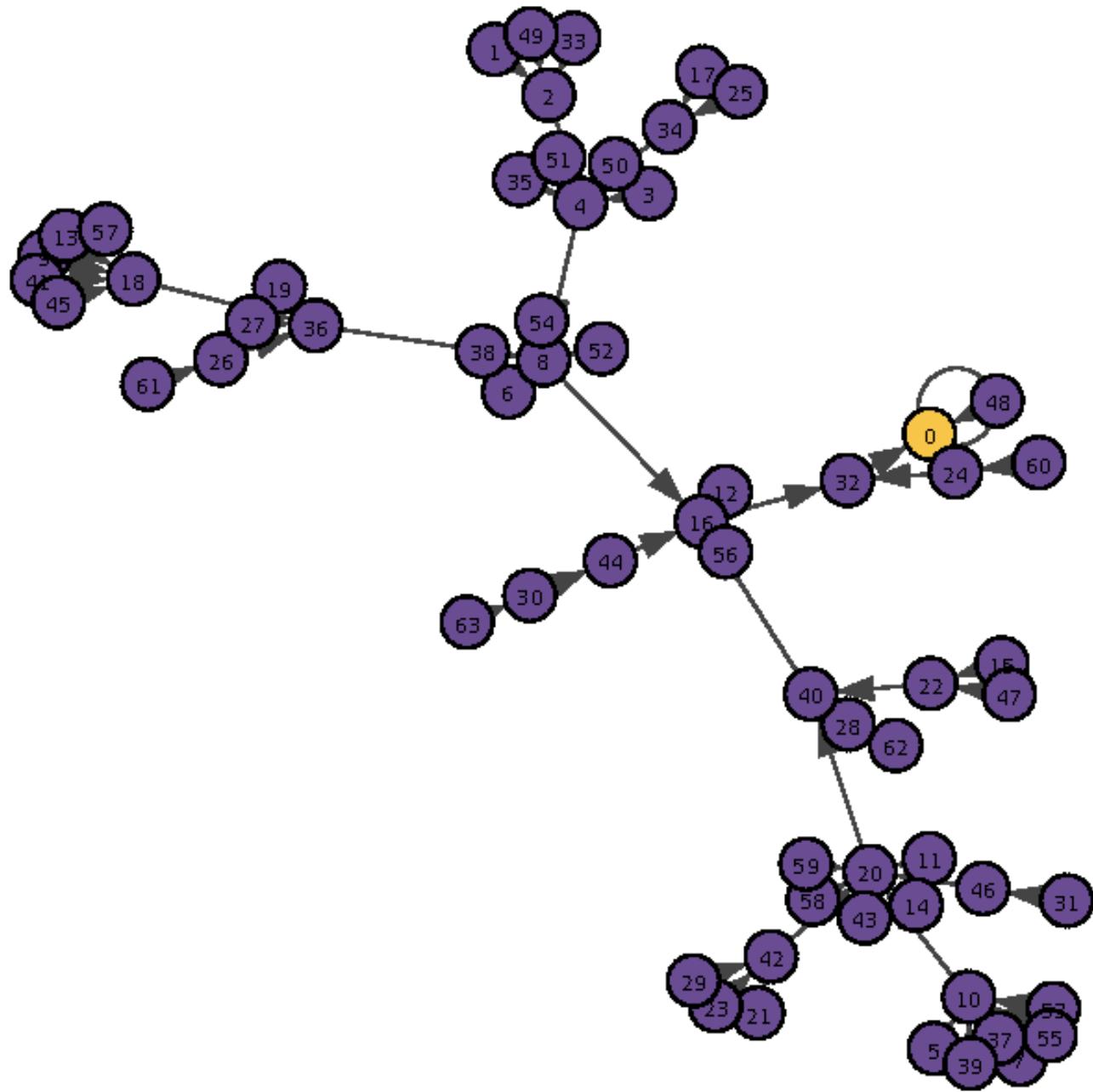


Figure 96: Grafo del atractor para la regla 162 con $n = 6$

3.1.80 Regla 164

Con un árbol principal del nodo 0 como atractor, se conforman otros árboles más pequeños de cadenas consecutivas y atractores marginales. Se observa una tendencia creciente en el número de árboles que se generan a medida que aumenta el valor de n .

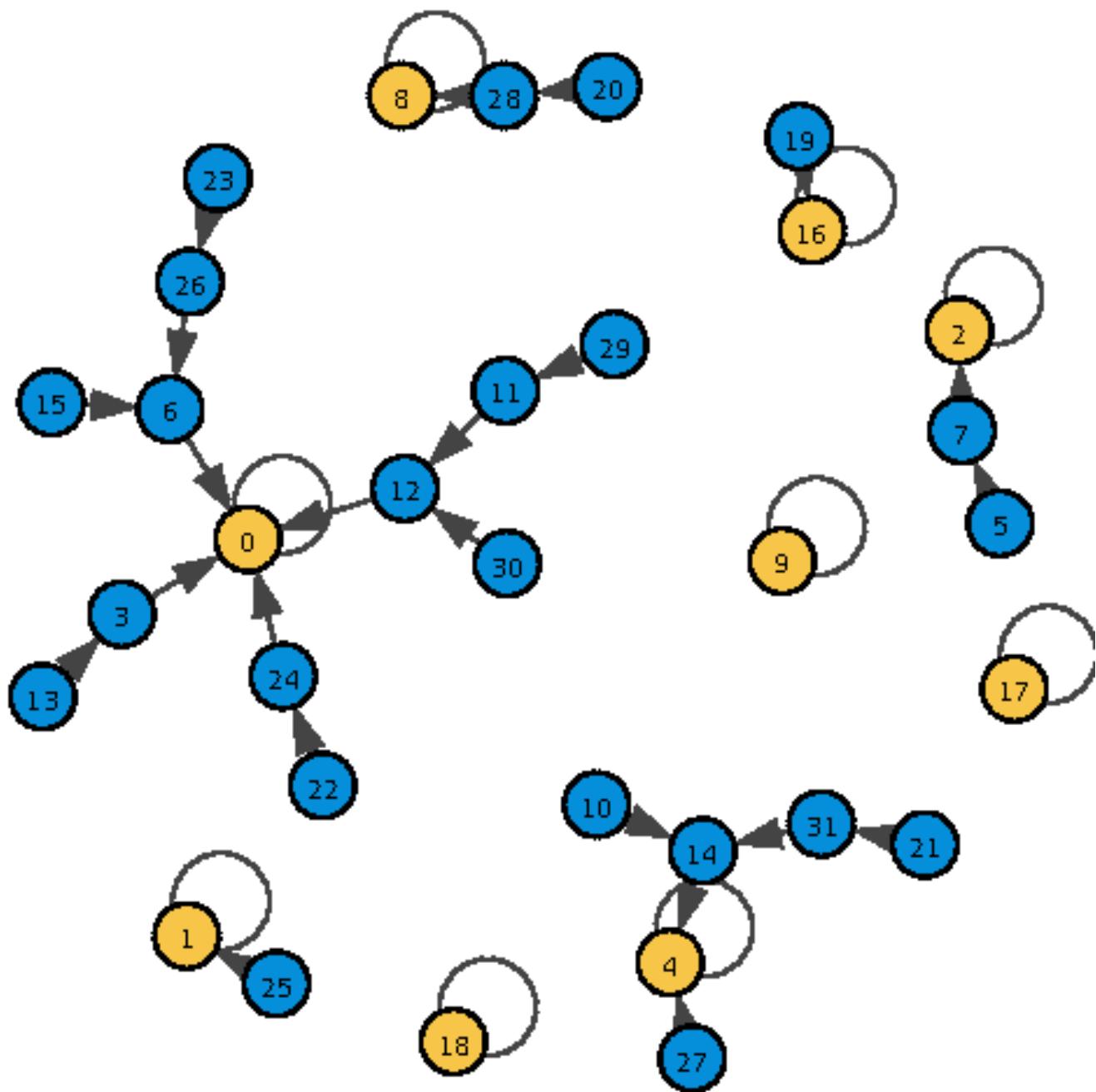


Figure 97: Grafo del atractor para la regla 164 con $n = 5$

3.1.81 Regla 168

Regla que genera un árbol con el nodo 0 como único atractor que muestra una rama principal con una cadena larga consecutiva de nodos, mientras más ramas se unen directamente al nodo atractor pero sobre las que los nodos extremos cuentan con capacidades de atracción conjuntando hojas del edén.

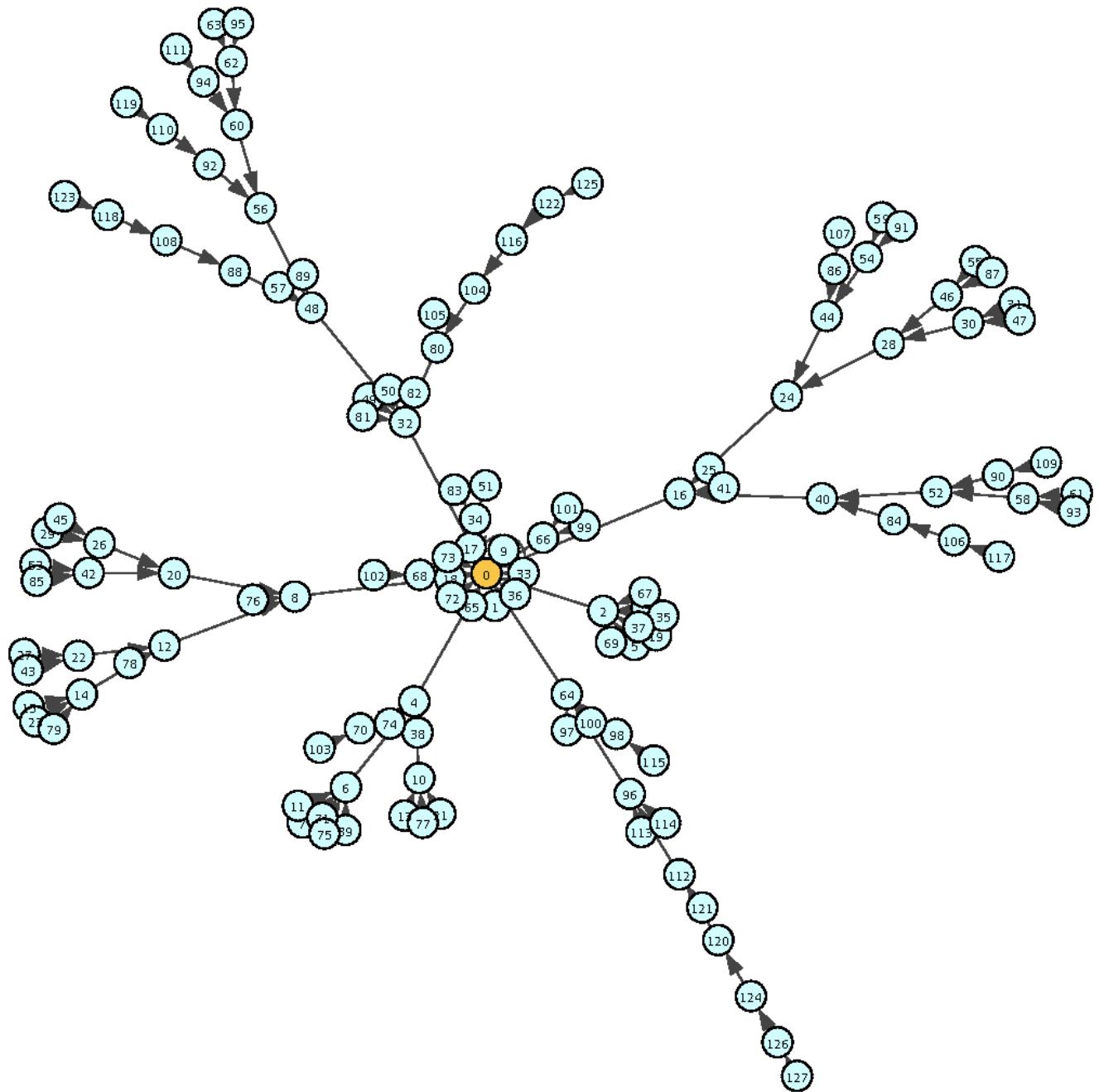


Figure 98: Grafo del atractor para la regla 168 con $n = 7$

3.1.82 Regla 170

Con características muy similares a reglas como la 162, 138, 56 y 34 entre otras, se genera un árbol único con el nodo 0 como atractor principal, con la particularidad de contar con ramas donde los nodos extremos cuentan con la propiedad atractora que les permite aglomerar nodos del edén como ancestros, y sin embargo el nodo atractor no cuenta con estas propiedades, teniendo un único ancestro.

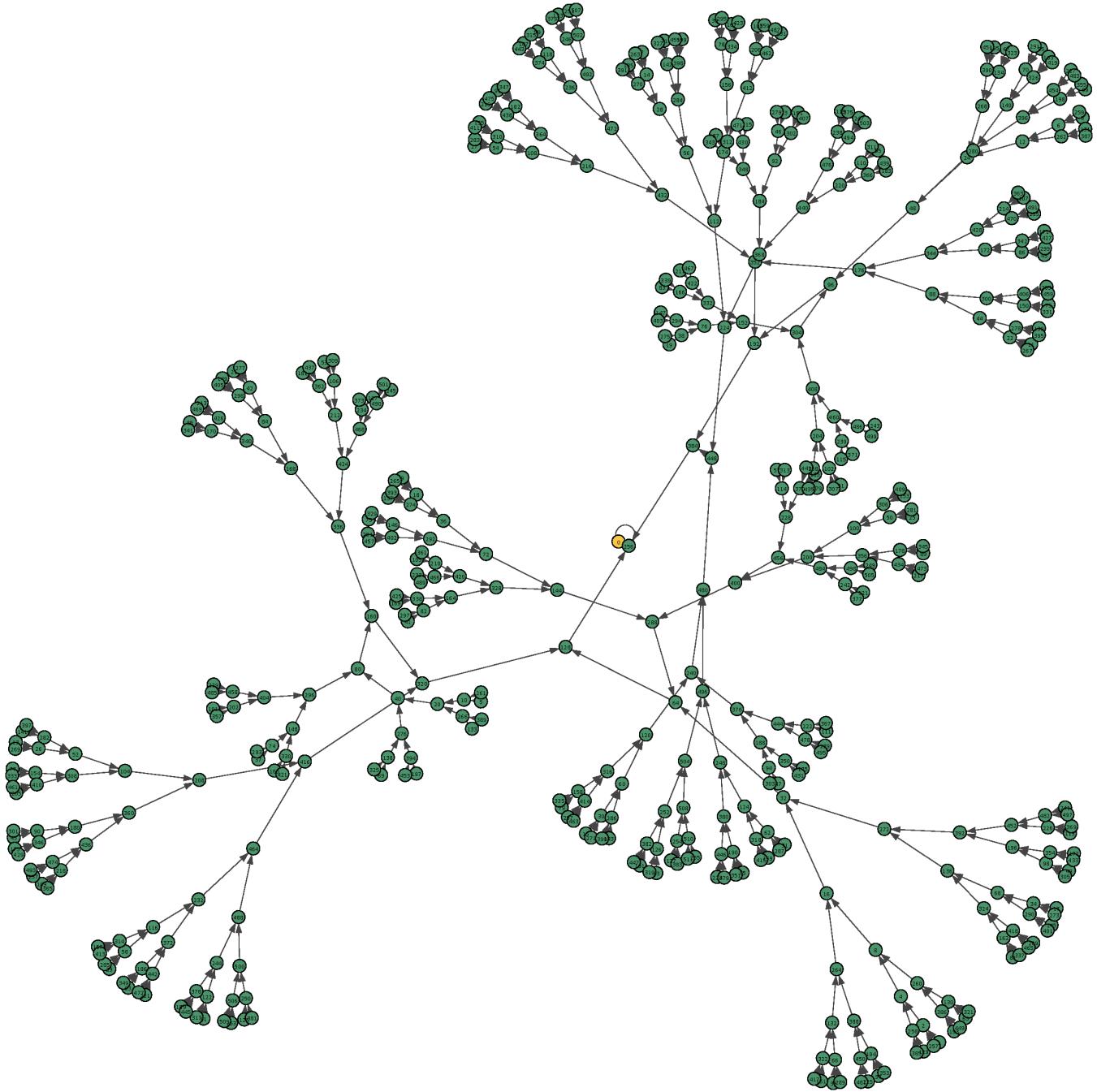


Figure 99: Grafo del atractor para la regla 170 con $n = 9$

3.1.83 Regla 172

Regla capaz de generar en números árboles de evolución, que particularmente para esta regla, comparten una rama principal extensa de nodos consecutivos, y de la cual se le agregan y derivan demás ramas de cadenas consecutivas. Se identifica una tendencia creciente a aumentar el número de estructuras, como así también la presencia de atractores marginales.

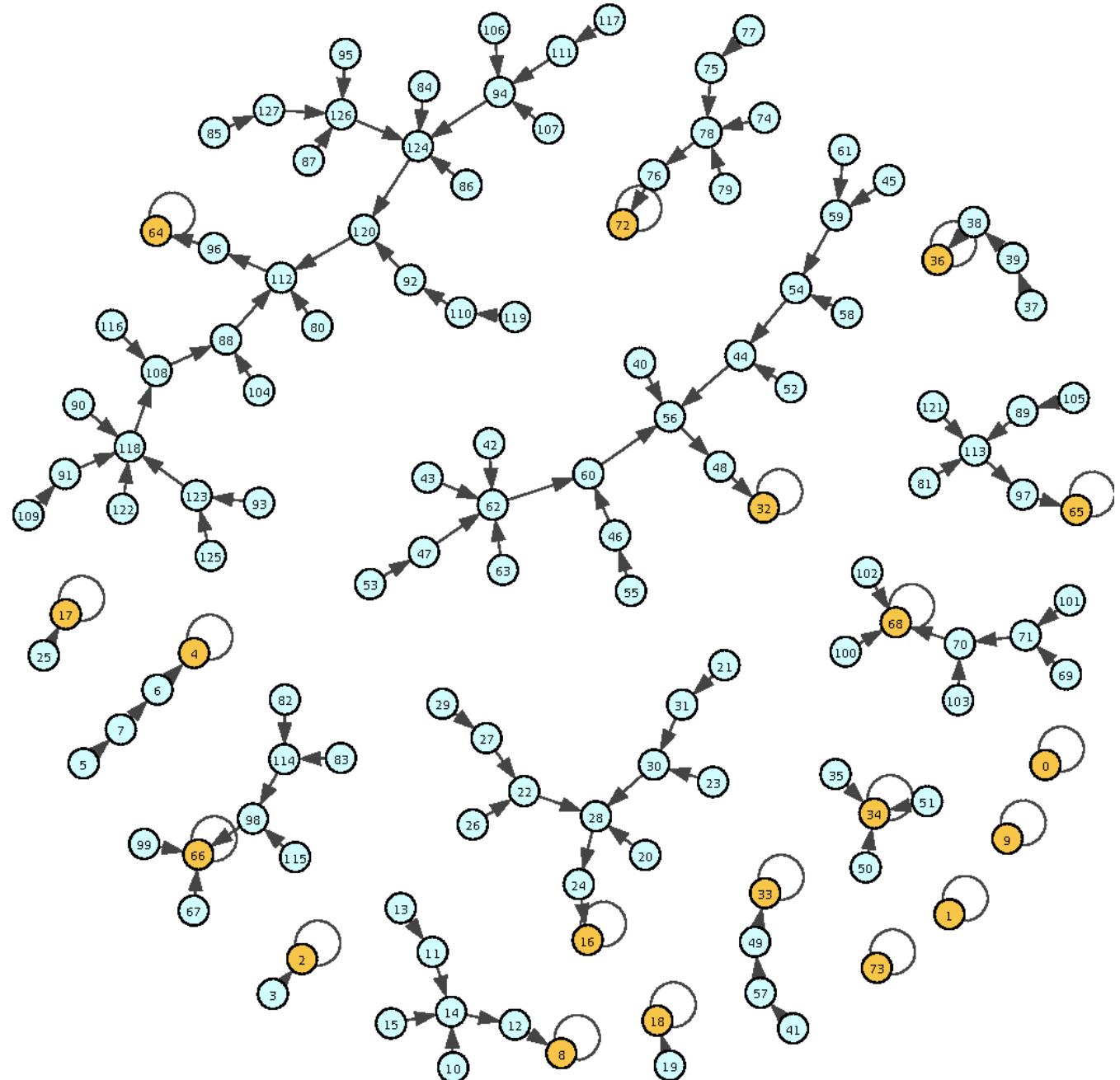


Figure 100: Grafo de los atractores para la regla 172 con $n = 7$

3.1.84 Regla 178

Conformando árboles con características nodos con propiedades atractoras, la regla 178 también genera algunas estructuras más pequeñas de árboles con cadenas consecutivas, sin embargo el nodo atractor principal acapara la mayor cantidad de nodos que lo conforman.

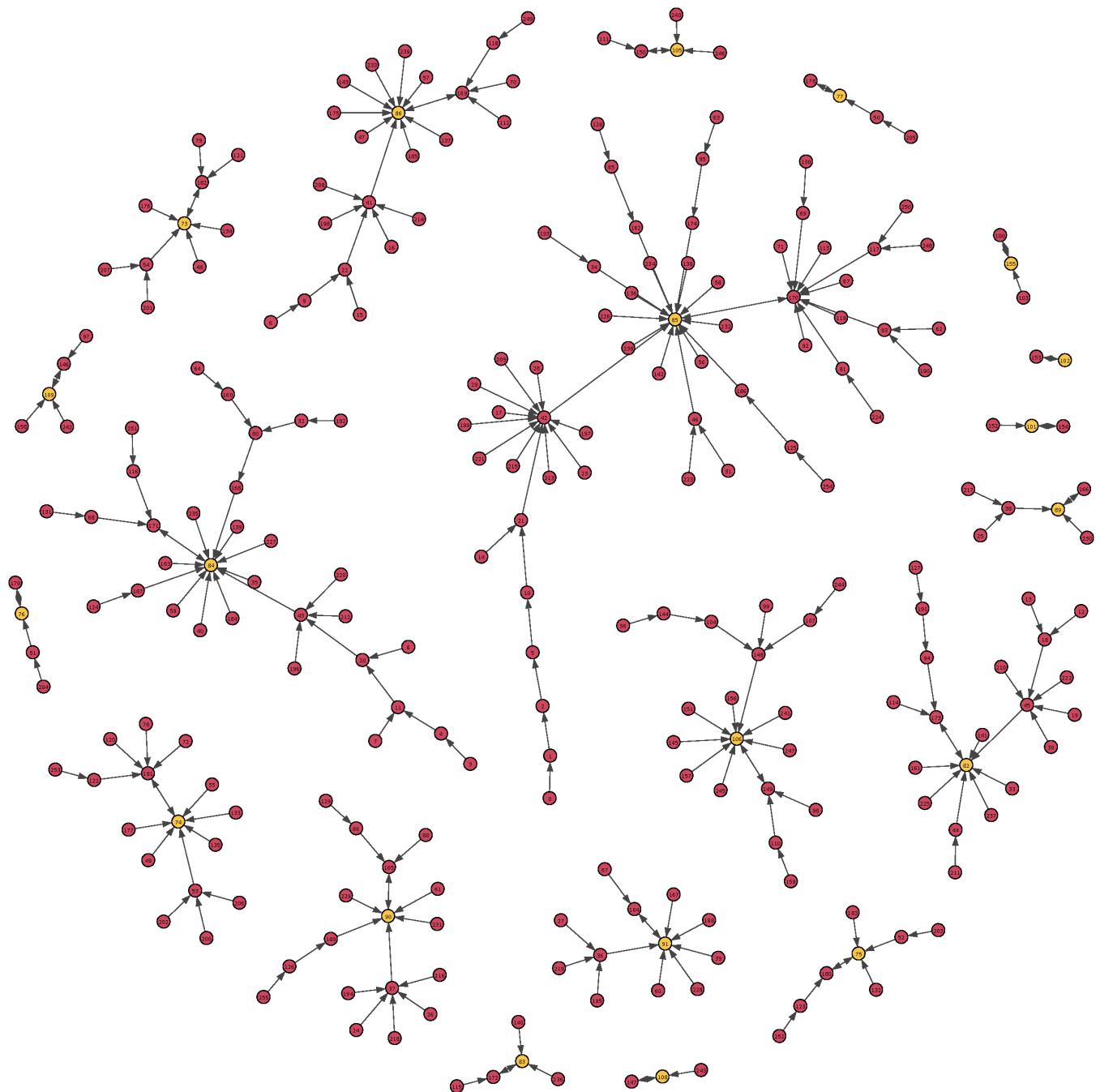


Figure 101: Grafo de los atractores para la regla 178 con $n = 8$

3.1.85 Regla 184

No muy distinto a reglas como la 162 y 138, se genera un árbol único con el nodo 0 como atractor principal, contando en los nodos de los extremos de las ramas propiedades atractoras. A pesar de esto se puede observar particular a esta regla, la asimetría con respecto al atractor, habiendo una concentración clara de mayor número de nodos hacia una de las 2 ramas que conectan al atractor.

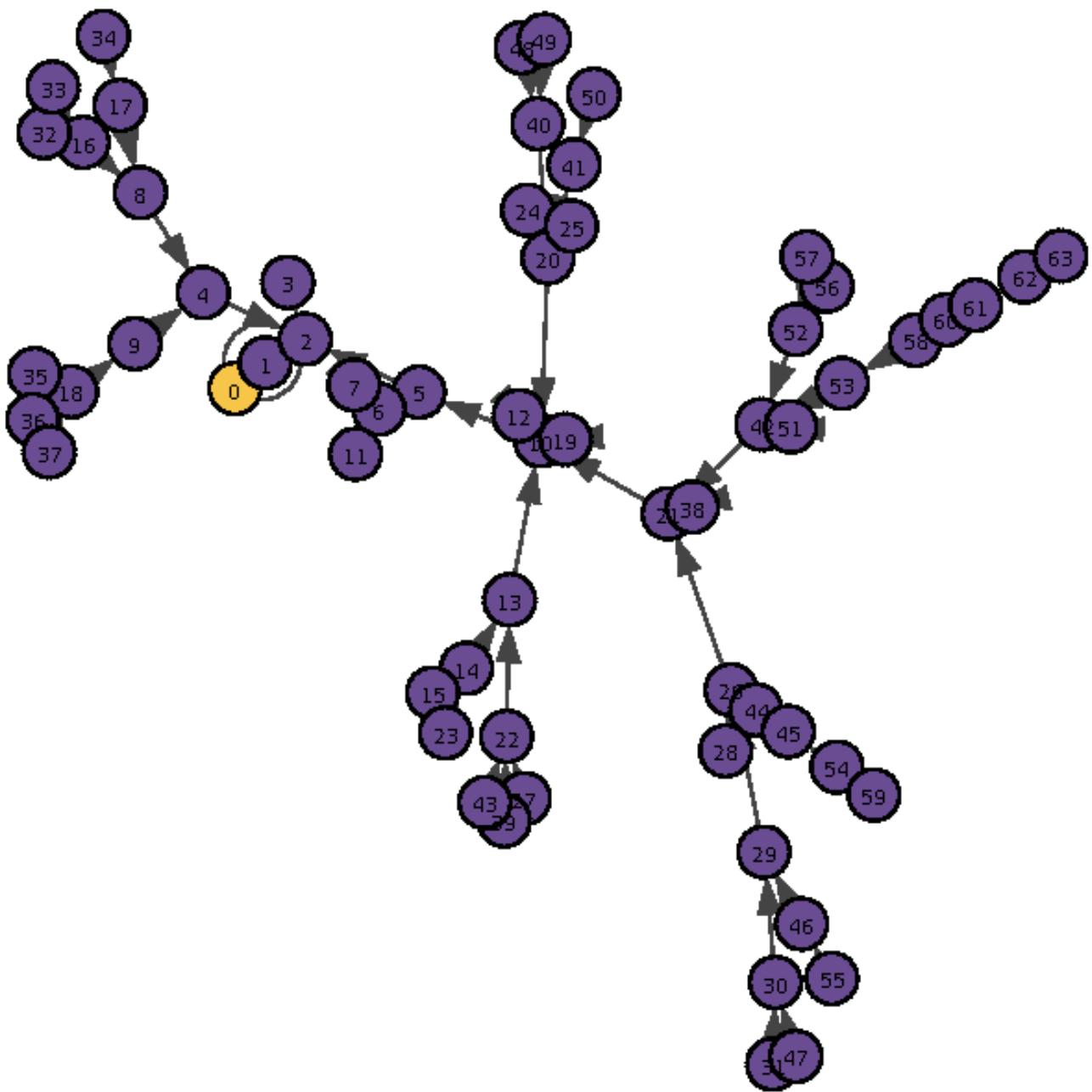


Figure 102: Grafo de los atractores para la regla 184 con $n = 6$

3.1.86 Regla 200

La regla 200 realiza construcciones características de campos de atracción con nodos atractores con fuertes campos de atracción, de esta forma se identifica una velocidad de convergencia de tan solo 1 evolución, encontrando a todos los nodos no atractores, como hojas del edén ancestros a un nodo atractor.

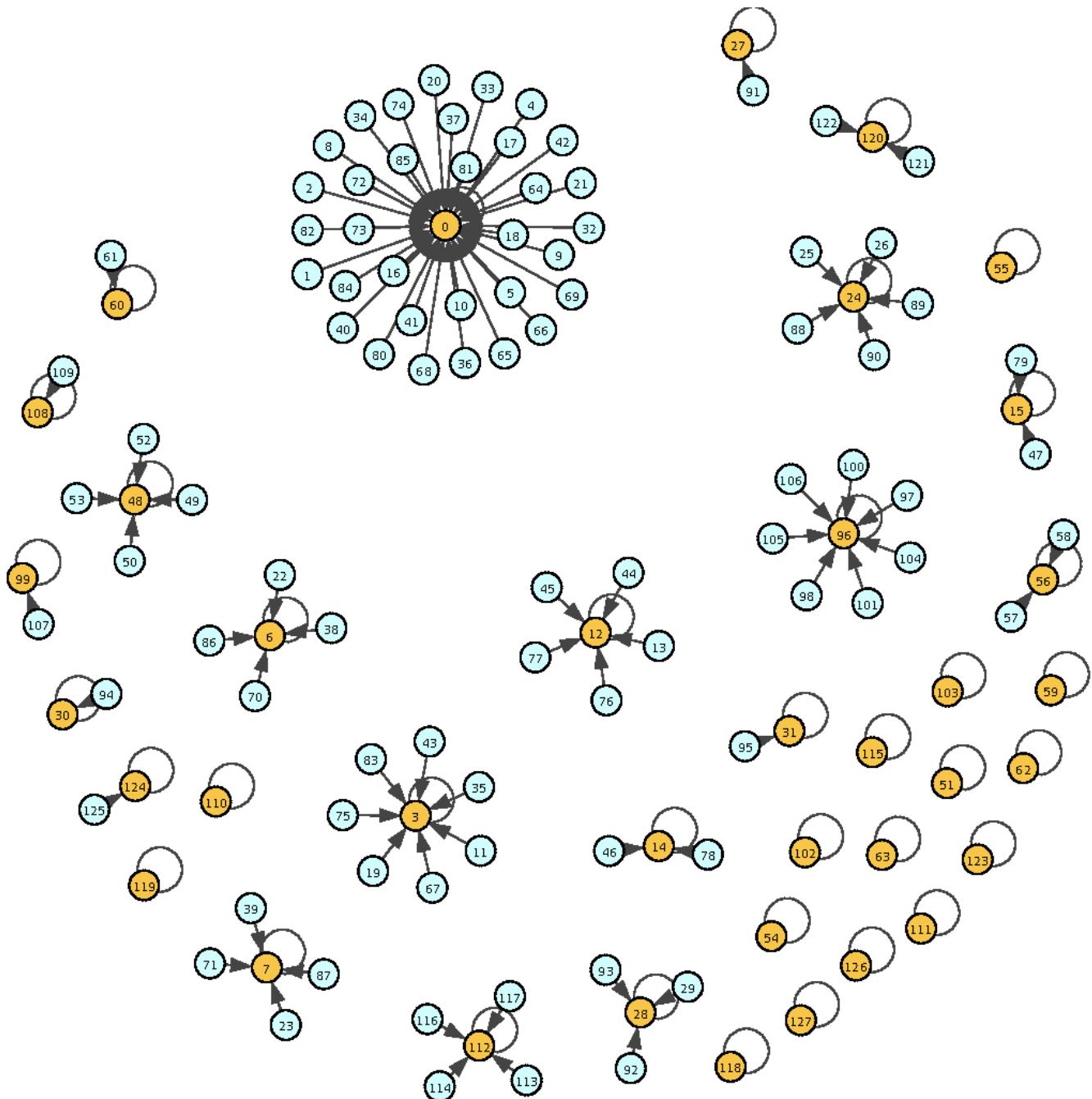


Figure 103: Grafo de los atractores para la regla 200 con $n = 7$

3.1.87 Regla 204

Regla muy característica donde para cada configuración posible en la evolución es un atractor, esto evidentemente asegura la convergencia inmediata de la evolución, eliminando completamente la complejidad del sistema y evitando la evolución en nuevas y emocionantes estructuras.

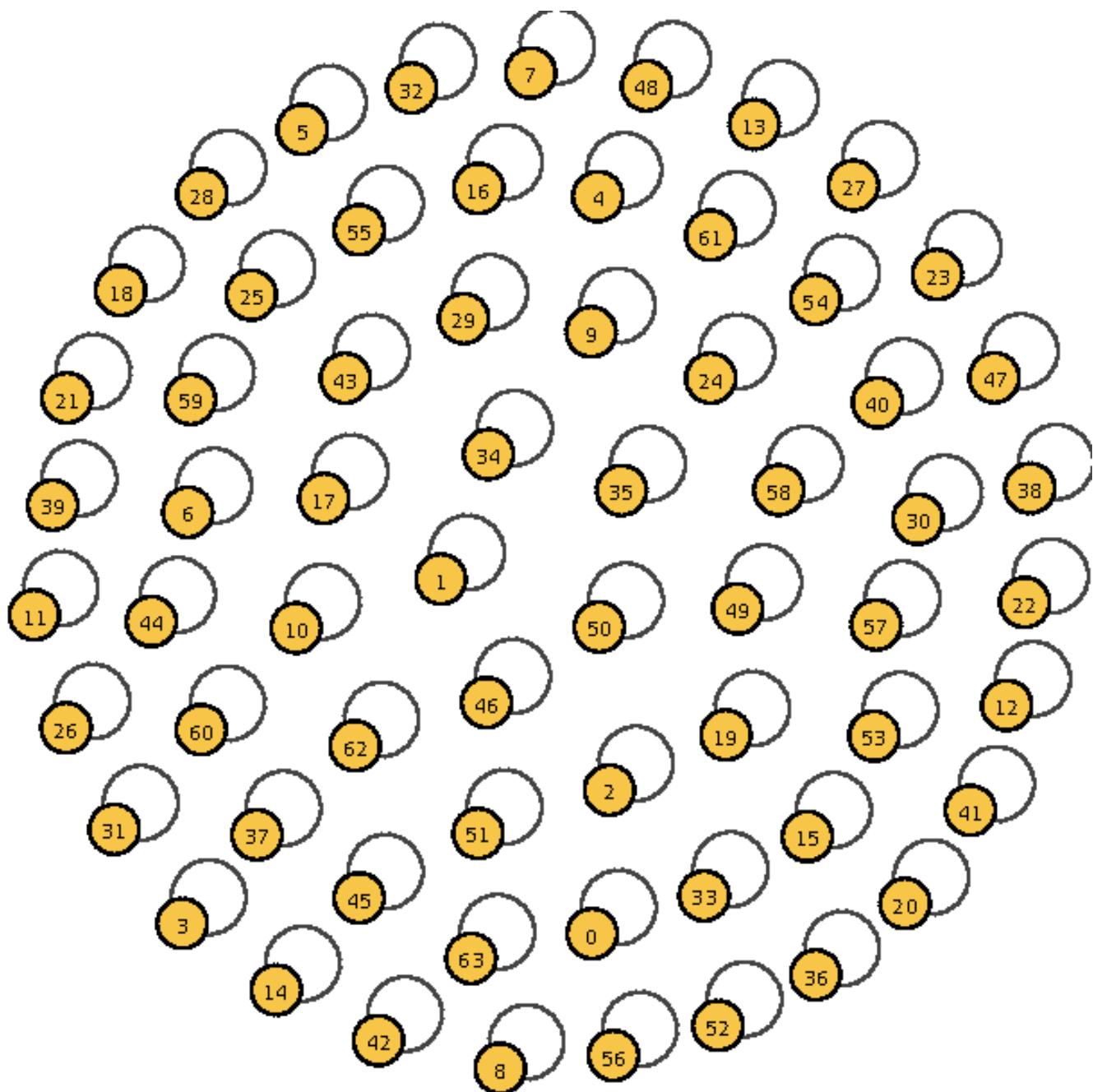


Figure 104: Grafo de los atractores para la regla 204 con $n = 7$

3.1.88 Regla 232

La regla 232 genera estructuras compactas resultado de los nodos atractores con campos fuertes de atracción, se tienen entonces una gran cantidad de hojas del edén como ancestros del atractores, lo que propicia a la estabilidad del sistema. Así también se advierten atractores marginales constantemente a través de las generaciones para diferentes n .

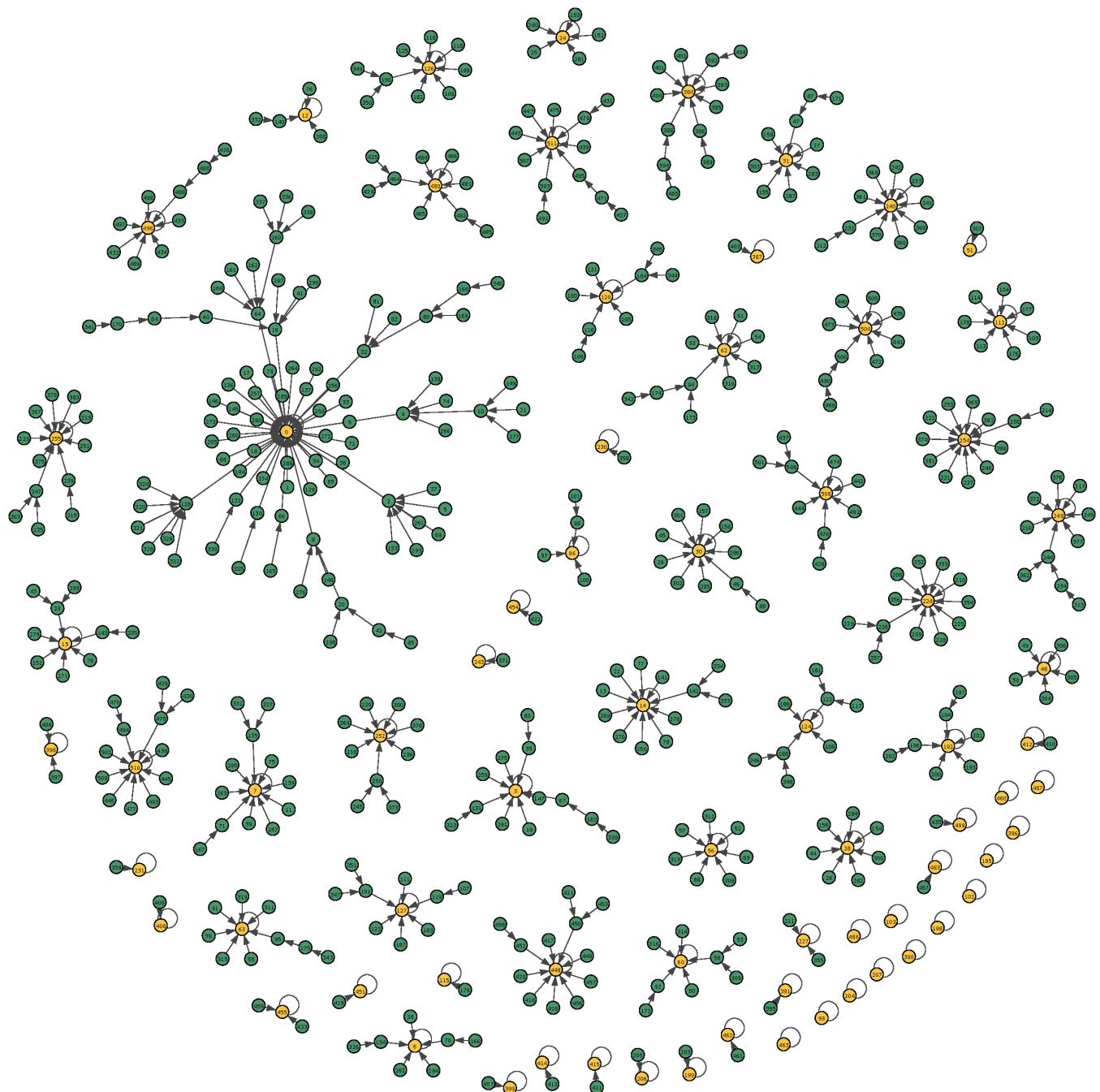


Figure 105: Grafo de los atractores para la regla 232 con $n = 9$

4 Conclusión

Con la elaboración de este segundo programa se tenían 2 objetivos. El primero de ellos la creación completa del simulador que permitiera observar mediante la animación gráfica, la evolución de las células unidimensionales según una regla especificada.

Como segundo objetivo se planteó el análisis de los atractores de las reglas, lo que requería del poder computacional y la infraestructura elaborada con el simulador de ECAs.

Habiendo aprendido del primer simulador bidimensional tipo *Life*, este segundo simulador se creó ocupando como base conceptual y gráfica, el primero, se evaluaron las áreas de oportunidad y se actualizaron según los requisitos de este segundo simulador.

Un aspecto en el que se basa y se le pone importante atención a este segundo simulador, es la optimización para la ejecución del aparto gráfico. Este aspecto muy diferente al instinto inicial que tuviera para su logro, cambió radicalmente los mecanismos de operación interna del simulador frente al primero.

Corriendo en un hilo único principal, el primer simulador realiza una reimpresión constante de la pantalla en una relación de 60 cuadros por segundo, situación que cuando se tenían espacios de evolución grandes de más de 500 células por dimensión, resultaba en una tarea pesada para el CPU por la cantidad de elementos a reimprimir además de la sobrecarga operacional que ocurría antes de realizar la impresión.

Considerando esto el enfoque que se tomó fue la creación de 3 hilos además del principal, el primero de ellos se encarga de atender los eventos desencadenados por las acciones de usuario: Clicks, presión de teclas, scrolling, etc. Que desencadenan entonces acciones que pueden modificar células, ingresar caracteres a los campos de texto, iniciar o pausar la evolución, etc. Procesos que son agregados a la cola de funciones.

Como segundo hilo se encarga de manejar los eventos internos que repercuten directamente a los gráficos de la simulación. Por el colapso del sprite en el cursor con un botón durante un click, una tecla presionada mientras el foco lo tiene un campo de texto, etc.

Y finalmente el tercer hilo enfocado en funciones, es capaz de ejecutar asíncronamente funciones de cualquier tipo, de esta forma si un proceso es demasiado lento se evita que el apartado gráfico también sufra las consecuencias, por lo que se simula una ejecución en segundo plano en una dinámica de cola de eventos.

Adicional a esta diversificación de los flujos de ejecución y atendiendo a las herramientas más técnicas de PyGame, se logra una gran optimización controlando específicamente cuando y que secciones de pantalla refrescar para mostrar en la GUI. Esto se logra gracias gracias a una combinación de los flujos alternos que permiten atender de manera asíncrona eventos internos, y la capacidad de PyGame de refrescar la pantalla en secciones cuando se le indique, permitiendo de esta forma que si en la interfaz no hay interacción con botones o cambios diversos en la interfaz, se evite completamente el proceso completo de reimpresión de todos los elementos, realizándose cuando existe una interacción o cambio, del componente específico que muestra un cambio visual.

Eso respecto a la infraestructura del simulador, ahora respecto a proceso de cálculo de atractores, se enfrentó a algunos problemas principalmente con la conversión de los grafos de entidades lógicas a imágenes para su interpretación. Como primer aproximación se ocupó un par de bibliotecas, *igraph* en conjunto con *cairocffi*, que para conjuntos relativamente pequeños de nodos, realizaban la imágenes de los grafos de manera excelente, sin embargo a partir de la potencia 2^{10} , no se logró conseguir pudieran visualizarse correctamente los grafos, atribuyéndose esta situación a la propia biblioteca que ya no es capaz de manejarlos, pues se intentó incrementado drásticamente el tamaño del lienzo a valores como $10k \times 10k$ pixeles, y aún con esto las distribuciones eran compactadas en grafos interpretables.

Se buscaron alternativas y se encontró la biblioteca *networkx*, la cual fue probada en el mismo rubro y era capaz de imprimir únicamente utilizando Matplotlib como herramienta subyacente, y de todas maneras creando distribuciones de los nodos para valores aún pequeños de potencia, completamente incorrectos y para nada interpretables.

Al final se decidió por utilizar *igraph* para la graficación limitada visualmente a la potencia 9, y *networkx* como herramienta para el análisis de los grafos, implementando herramientas sumamente útiles que permitió identificar los atractores de cada árbol de evolución sin una inspección humana previa, y esto facilitó la asignación de un color(amarillo) distinto a este nodo como ayuda visual en la imagen de las evoluciones.

Ambos objetivos se lograron y se cubrieron todos los requerimientos especificados tanto para el programa simulador, como para el análisis completo de atractores, enfrentando por su puesto algunas complicaciones de las cuales la gran mayoría fueron atendidas y subsanadas para entregar un simulador completamente funcional que además incluye la función para la identificación de los campos atractores de una regla en un rango de potencias.

5 Código Fuente

El código fuente también puede ser encontrado en el repositorio de GitHub para su consulta o descarga para prueba: https://github.com/LaloValle/Simulador_ECAs_y_atractores

El código fuente está conformado por un total de 5 módulos de Python y un archivo principal donde se cuenta con la función main que controla el flujo principal del programa y contexto gráfico para la biblioteca PyGame.

5.1 Módulos

Los módulos son archivos de código Python que agrupan clases, funciones, constantes, etc. que semánticamente se encuentran correlacionadas bajo un contexto común. Los siguientes módulos contienen en su mayoría Clases que definen componentes lógicos, nodos de un sistema, componentes gráficos; también pueden contener Constantes de uso extendido entre los módulos del programa.

5.1.1 Constant

Módulo que integra algunos recursos constantes comunes a varios de los módulos en la simulación. Los recursos incluyen tuplas de colores en formato RGB, instancias de fuentes para PyGame y matrices de conversión.

```
import pygame
from numpy import array

=====
# Colors
=====
DARK_BLACK = (28,29,32)
LIGHT_BLACK_1 = (40,44,52)
LIGHT_BLACK_2 = (53,61,76)
# Gray pair
WHITE = (192,203,221)
GRAY = (162,170,185)
# Blue pair
BLUE = (17,192,186)
LIGHT_BLUE = (168,234,232)
# LIGHT_BLUE = (79,189,186)
# Yellow pair
YELLOW = (234,194,76)
LIGHT_YELLOW = (234,213,150)
# Red pair
RED = (206,71,96)
LIGHT_RED = (234,159,175)
# Green pair
GREEN = (0,161,157)
LIGHT_GREEN = (79,189,186)

COLORS_LIST =
    [DARK_BLACK,LIGHT_BLACK_1,LIGHT_BLACK_2,WHITE,GRAY,BLUE,LIGHT_BLUE,YELLOW,LIGHT_YELLOW,RED,LIGHT_RED,GREEN,LIGHT_GR
HEX_COLORS_LIST = [
    '#D2FDFF', # Light Cyan
    '#CE4760', # Brick Red
    '#4C956C', # Middle Green
    '#eeeebd3', # White
    '#058ED9', # Green Blue Crayola
    '#6a4c93', # Royal Purple
]
COLOUR_ATTRACTOR = '#F7C548'
```

```

#=====
# Bits mask for number of neighbourhood in the ECA
#=====

BITS_MASKS = [
    1, # 0 Neighbourhood
    2, # 1 Neighbourhood
    4, # 2 Neighbourhood
    8, # 3 Neighbourhood
    16, # 4 Neighbourhood
    32, # 5 Neighbourhood
    64, # 6 Neighbourhood
    128, # 7 Neighbourhood
]
#=====
# Weighted matrix for binary to decimal conversion in ECA
#=====

MATRIX_BIN_TO_DEC = array([4, 2, 1])

#=====
# Fonts
#=====

pygame.font.init()
FONT = pygame.font.SysFont('Silka', 15, False, False)
SMALL_FONT = pygame.font.SysFont('Silka', 12, False, False)
MEDIUM_FONT = pygame.font.SysFont('Silka', 25, False, False)
BIG_FONT = pygame.font.SysFont('Silka', 50, False, False)

```

5.1.2 Layouts

Parte de los módulos relacionados a los gráficos, este módulo está conformado por clases que como elementos utilizados en interfaces gráficas, funcionan como herramientas que dan estructura a los elementos o que aportan cierta funcionalidad relacionada con su estructuración. Estos elementos pueden tener, o no, una representación gráfica visual.

Clase Grid Mencionada en secciones anteriores, esta clase define un componente estructural sin representación visual, pero que provee métodos útiles para la estructuración automática de elementos en un formato tipo tabla.

Clase SideBar Elemento gráfico estructural con representación visual como de un gran rectángulo que abarca lo alto de la ventana, es utilizada para agrupar visualmente elementos gráficos, y proveer una posición de referencia para colocar otros componentes. En particular esta clase es una clase *Singleton*, de forma que en el contexto de la simulación existe una instancia única, por lo que la definición y colocación de elementos como botones, etc. Se realiza directamente en la clase.

Clase BottomBar Elemento gráfico estructural con representación visual como un delgado rectángulo ubicado en la parte inferior de la ventana ocupando todo su ancho. Con comportamiento parecido a la clase SideBar, sirve como agrupador visual y como referencia para otros componentes, siendo en este caso el de los textos que indican las variables dinámicas de la simulación.

```

import pygame
import numpy as np
# User modules
import Graphics as grph
from Constants import COLORS_LIST, FONT
from GraphicalComponents import *

class Grid():
    """Class that defines a grid given a position, number of columns, rows and a padding
    Helps to locate easily a list of elements in a grid layout

    Functions
    -----
    optimal_grid_size( grid_sizes, num_cols, num_rows, number_elements, padding ): optimal_grid_sizes,
        elements_size
        Class function that returns an array with the adjusted real size of a grid given the

```

number of elements to be allocated and the padding for a number of columns and rows.
The last return is a new instance of the Grid class with the given and calculated parameters

Methods

```
get_element_size(self) : self.element_size
    Getter for the size of the elements in the grid

calculate_element_size(self) :
    Method that computes the size of each element depending on the number of columns
    and padding specified when creating the grid

locate_elements(self,list_elements) :
    Locates the elements provided in the grid from left to right, and top to bottom
"""

#-----
# Class functions
#-----
def optimal_grid_size( grid_sizes:tuple, num_cols:int, num_rows:int, padding:int=5, use_min:bool=False ):
    grid_sizes = np.array(grid_sizes,np.uint32)
    # Space of the paddings for all the grid in each column and row
    paddings = np.array([padding*(num_cols+1),padding*(num_rows+1)],np.uint32)
    elements_sizes = np.around( (grid_sizes - paddings) / # The original size of the grid gets substracted
        the number of paddins by row and column
        np.array([num_cols,num_rows],np.uint32)) # The resultant array gets divided between the
        number of columns and rows respectively
    elements_sizes[:] = elements_sizes.min() if use_min else elements_sizes.max()
    optimal_grid_sizes = (elements_sizes*np.array([num_cols,num_rows],np.uint32) +
        paddings).astype('uint32')
    # Creates a new instance of the Grid class with the parameters specified
    new_grid_instance =
        Grid(optimal_grid_sizes,num_cols=num_cols,num_rows=num_rows,padding=padding,element_sizes=tuple(elements_size))
    return optimal_grid_sizes,new_grid_instance

def __init__(self, size:np.array, coord:tuple=(0,0), num_cols:int=2, num_rows:int=2, padding:int=5,
    element_sizes:tuple=(-1,-1), window=None):
    self.window = window
    self.size = size
    self.coord = coord
    self.num_cols = num_cols
    self.num_rows = num_rows
    self.padding = padding
    self.element_sizes = self.calculate_element_sizes() if element_sizes[0]+element_sizes[1] < 0 else
        element_sizes
#-----
# Getters and setters
#-----
def set_window(self,window):
    self.window = window

def get_element_sizes(self):
    return self.element_sizes
#-----
# Grid Methods
#-----
def calculate_element_sizes(self):
    """Computes the size of each element if not given when instanciating

Returns
-----
element_sizes : tuple
    Tuple with the sizes of width and height
"""

```

```

        return tuple(np.around(
            (self.size -
             np.array([self.padding*(self.num_cols+1),self.padding*(self.num_rows+1)],np.uint16))
            / # The original size of the grid gets subtracted the number of paddins by row and
            # column
             np.array([self.num_cols,self.num_rows],np.uint16)) # The resultant array gets divided
            # between the number of columns and rows respectively
        )
    )

def locate_elements(self,list_elements):
    """Method that assigns the coordinates of the corner-top-left of each element in the list
    so that they are arranged in the grid previously calculated

    Parameters
    -----
    list_elements : list
        List with the 'rects' of each graphical element that allows the element to be moved to
        a given coordinate
    """
    y_pos = self.padding+self.coord[1] # Initial position with just the padding added to the original y
    # Loop in number of rows
    for row in range(self.num_rows):
        x_pos = self.padding+self.coord[0] # Initial position with just the padding added to the oiriginal x
        # Loop in number of columns
        for column in range(self.num_cols):
            index = row*self.num_cols+column
            if index < len(list_elements): list_elements[index].move((int(x_pos),int(y_pos)))
            x_pos += self.padding+self.element_sizes[0] # The next element must be placed in the x position
            # with added padding and width of the previous element
        y_pos += self.padding+self.element_sizes[1] # The next row of elements must be placed in the y
        # position with added padding and width of the previous elements
    """

class BottomBar():
    """Simple rectangle used as bottom bar
    As there is only 1 bottom bar in the program, a Singleton Class is implemented
    so it can be retreived in any part of the code with the configured values for
    the attributes and graphical components inside it

    Methods
    -----
    draw(self,window) :
        Draws a rectagle at the bottom with a given height and background color

    update_generations(self,generations) :
        Updates the generations string to be printed

    update_alive_cels(self,alive_cells) :
        Updates the alive cells string to be printed

    update_space_dimension_zoom(self,alive_cells) :
        Updates the dimensions string of the grid and the zoom string
    """

    # Singleton Class
    bottom_bar = None

    def __init__(self,color_background:tuple,height:int=35,number_rows:int=100,number_cells_1D:int=100):
        BottomBar.bottom_bar = self

        self.color_background = color_background
        self.height = height

```

```

# The position and size gets define according to the windows size
window_size = pygame.display.get_window_size()
self.rectangle = pygame.Rect(
    ( 0, window_size[1]-height ),
    ( window_size[0]-250, height )
)

# Creation of texts
self.texts = []
self.generations_text = Text((10,3+self.rectangle.y), 'Generations:')
self.alive_cells_text = Text((self.rectangle.width-30-FONT.size('Alive Cells: 0')[0],
    3+self.rectangle.y), 'Alive Cells: 0')
self.dimensions_number_space_text = Text((self.rectangle.width/2-int(FONT.size('100x100(space
    1)')[0]/2), 3+self.rectangle.y), '{}x{}'.format(number_cells_1D,number_rows))
self.texts.append(self.generations_text)
self.texts.append(self.alive_cells_text)
self.texts.append(self.dimensions_number_space_text)

#-----
# Drawing method
#-----
def draw(self,window):
    pygame.draw.rect(window,self.color_background,self.rectangle)
    # Prints the texts strings
    for txt in self.texts:
        txt.print(window)
#-----
# Updates of strings
#-----
def update_generations(self,generations):
    self.generations_text.update('Generations:' + str(generations))
    return self.generations_text.rect

def update_alive_cells(self,alive_cells):
    self.alive_cells_text.update('Alive Cells: ' + str(alive_cells))
    return self.alive_cells_text.rect

def update_dimensions_number_space(self,space_dimension,number_space):
    self.dimensions_number_space_text.update('{}x{}'.format(space_dimension[1],space_dimension[0],number_space))
    return self.dimensions_number_space_text.rect

class SideBar():
    """Simple rectangle used as side bar
    As there is only 1 side bar in the program, a Singleton Class is implemented
    so it can be retrieved in any part of the code with the configured values for
    the attributes and graphical components inside it

    Methods
    -----
    draw(self,window) :
        Draws a rectagle at the right side with a given width and background color
    """
    #
    # Singleton Class
    #
    side_bar = None

    #
    # Button constants of class
    #

```

```

PLAY_BUTTON = 0
STOP_BUTTON = 1
RESTART_BUTTON = 2
CLEAR_BUTTON = 3
SAVE_BUTTON = 4
UPLOAD_BUTTON = 5
RANDOM_INITIAL_BUTTON = 6
INITIAL_CENTER_CELL_BUTTON = 7
DRAG_SLIDER_BUTTON = 8
RULE_INPUT = 9
DENSITY_BUTTON = 10
LOGARITHM_BUTTON = 11
ENTROPY_BUTTON = 12
ALIVE_COLOR_INPUT = 13
DEAD_COLOR_INPUT = 14
ATTRACTORS_BUTTON = 15
START_ATTRACTORS_INPUT = 16
END_ATTRACTORS_INPUT = 17

def __init__(self,color_background:tuple,width:int=250,bottom_margin:int=0):
    # The only instance is the first created in the main
    SideBar.side_bar = self

    self.color_background = color_background
    self.width = width
    self.bottom_margin = bottom_margin
    self.padding = 15

    # The position and size gets define according to the windows size
    window_size = pygame.display.get_window_size()
    self.rectangle = pygame.Rect(
        (window_size[0]-width, 0),
        (width, window_size[1]-bottom_margin)
    )

    # Graphical elements inside the bar
    self.graphical_elements = []
    # Sections
    self.configurations_section =
        Section(self.rectangle.x,self.padding*2+80+10,self.rectangle.width,self.padding,'RUNNING
        CONFIGURATIONS')
    self.plot_section = Section(self.rectangle.x,270,self.rectangle.width,self.padding,'PLOTTING')
    self.colors_section =
        Section(self.rectangle.x,375+self.padding,self.rectangle.width,self.padding,'COLORS')
    self.attractors_section =
        Section(self.rectangle.x,480+self.padding,self.rectangle.width,self.padding,'ATTRACTORS')
    self.graphical_elements.append(self.plot_section)
    self.graphical_elements.append(self.configurations_section)
    self.graphical_elements.append(self.colors_section)
    self.graphical_elements.append(self.attractors_section)
    # Slider
    self.slider = self.set_slider()
    self.graphical_elements.append(self.slider)
    # Graphical sprites inside the bar
    self.graphical_sprites = pygame.sprite.Group()
    self.graphical_sprites.add(self.set_play_button())
    self.graphical_sprites.add(self.set_stop_button())
    self.graphical_sprites.add(self.set_restart_button())
    self.graphical_sprites.add(self.set_clear_button())
    self.graphical_sprites.add(self.set_save_button())
    self.graphical_sprites.add(self.set_upload_button())
    self.graphical_sprites.add(self.set_random_initial_button())
    self.graphical_sprites.add(self.set_initial_center_cell_button())

```

```

# Configurations elements
self.graphical_sprites.add(self.slider.get_drag_button())
self.graphical_sprites.add(self.set_rule_input())
# Plotting buttons
self.graphical_sprites.add(self.set_density_button())
self.graphical_sprites.add(self.set_logarithm_button())
self.graphical_sprites.add(self.set_entropy_button())
# Colors elements
self.graphical_sprites.add(self.set_alive_color_input())
self.graphical_sprites.add(self.set_dead_color_input())
# Attractors elements
self.graphical_sprites.add(self.set_attractors_button())
self.graphical_sprites.add(self.set_start_attractors_input())
self.graphical_sprites.add(self.set_end_attractors_input())

#
# Getters and setters
#
def get_graphical_sprites(self):
    return self.graphical_sprites

def set_click_function(self,button,function):
    """Sets the function to be actioned when the button gets pressed

Parameters
-----
button : int
    Index of the button in the graphical_sprites Sprite Group
    Class constants are used to describe the correct index for each button

function : function
    Reference to the function to be executed when the button gets clicked
"""
    self.graphical_sprites.sprites()[button].click_function = function

def get_sprite(self,index_sprite):
    return self.graphical_sprites.sprites()[index_sprite]

#
# Configuration of buttons
#
def set_play_button(self):
    window_size = pygame.display.get_window_size()
    # Size of the button
    radius = 20
    image_size = 25
    # The position in the center horizontally and at the top
    x = window_size[0] - self.width/2 - radius
    y = self.padding + radius + 5
    return CircularButton(x,y,radius,image_size,image_path='./images/play_w.png')

def set_stop_button(self):
    window_size = pygame.display.get_window_size()
    # Size of the button
    radius = 20
    image_size = 15
    x = window_size[0] - self.width/2 - 2*radius - 15*2
    y = self.padding + radius + 5

    return CircularButton(x,y,radius,image_size,image_path='./images/stop_w.png')

def set_restart_button(self):
    window_size = pygame.display.get_window_size()
    # Size of the button
    radius = 20

```

```

image_size = 20
x = window_size[0] - self.width/2 + 1.5*radius
y = self.padding + radius + 5

return CircularButton(x,y,radius,image_size,image_path='./images/restart_w.png')

def set_clear_button(self):
    window_size = pygame.display.get_window_size()
    # Size of the button
    radius = 18
    image_size = 13
    x = window_size[0] - 2*radius - self.padding
    y = self.rectangle.height - 2*radius - 2*self.padding

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/clear_w.png')

def set_save_button(self):
    window_size = pygame.display.get_window_size()
    # Size of the button
    radius = 18
    image_size = 13
    x = window_size[0] - 4*radius - 8 - self.padding
    y = self.rectangle.height - 2*radius - 2*self.padding

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/save_w.png')

def set_upload_button(self):
    window_size = pygame.display.get_window_size()
    # Size of the button
    radius = 18
    image_size = 13
    x = window_size[0] - 6*radius - 16 - self.padding
    y = self.rectangle.height - 2*radius - 2*self.padding

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/upload_w.png')

def set_random_initial_button(self):
    window_size = pygame.display.get_window_size()
    # Size of the button
    radius = 18
    image_size = 18
    x = window_size[0] - 2*radius - 8
    y = self.configurations_section.header_rect.y - 2*radius - 5

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/random_2_w.png')

def set_initial_center_cell_button(self):
    window_size = pygame.display.get_window_size()
    # Size of the button
    radius = 18
    image_size = 18
    x = window_size[0] - 2*radius - 8
    y = self.configurations_section.header_rect.y - 4*radius - 5

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/between_w.png')

def set_density_button(self):
    # Size of the button
    radius = 30
    image_size = 30
    x = self.rectangle.x + self.padding
    y = self.plot_section.header_rect.y + 30 + self.padding

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/density_w.png')

```

```

def set_logarithm_button(self):
    # Size of the button
    radius = 30
    image_size = 30
    x = self.rectangle.x + self.padding + 8 + 2*radius
    y = self.plot_section.header_rect.y + 30 + self.padding

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/logarithm_w.png')

def set_entropy_button(self):
    # Size of the button
    radius = 30
    image_size = 30
    x = self.rectangle.x + self.padding + 16 + 4*radius
    y = self.plot_section.header_rect.y + 30 + self.padding

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/entropy_w.png')

def set_slider(self):
    width = self.rectangle.width - 2*self.padding - 10
    x = self.rectangle.x + self.padding + 5
    y = self.configurations_section.header_rect.y + 30 + self.padding

    slider = Slider(x,y,width)
    slider.drag_button.click_function = self.slider_move
    return slider

def set_rule_input(self):
    x = self.rectangle.x + self.padding + 5
    y = self.configurations_section.header_rect.y + 30 + self.padding + 60

    input = Input(x,y,30,label='Rule',value='110',allow_focus=True)
    input.set_click_function(input.set_on_focus)
    return input

def set_alive_color_input(self):
    x = self.rectangle.x + self.padding + 5
    y = self.colors_section.header_rect.y + 30 + self.padding
    background_color = grph.Graphics.cells_colors[1]
    input =
        Input(x,y,45,label='Alive',width=FONT.size('Alive')[0]+50,allow_focus=False,background_color=background_color)
    input.set_click_function(self.change_alive_cell_color)
    return input

def set_dead_color_input(self):
    x = self.rectangle.x + 2*self.padding + 5 + 95
    y = self.colors_section.header_rect.y + 30 + self.padding
    background_color = grph.Graphics.cells_colors[0]
    input =
        Input(x,y,45,label='Dead',width=FONT.size('Dead')[0]+50,allow_focus=False,background_color=background_color)
    input.set_click_function(self.change_dead_cell_color)
    return input

def set_attractors_button(self):
    # Size of the button
    radius = 30
    image_size = 30
    x = self.rectangle.x + self.padding
    y = self.attractors_section.header_rect.y + 30 + self.padding

    return CircularButton(x,y,radius,image_size,border=False,image_path='./images/attractors_w.png')

def set_start_attractors_input(self):
    x = self.rectangle.x + 2*self.padding + 60
    y = self.attractors_section.header_rect.y + 30 + self.padding

```

```

    input = Input(x,y,30,width=135,label='Start',value='1',allow_focus=True)
    input.set_click_function(input.set_on_focus)
    return input

def set_end_attractors_input(self):
    x = self.rectangle.x + 2*self.padding + 60
    y = self.attractors_section.header_rect.y + 30 + self.padding + 30

    input = Input(x,y,30,width=135,label='End',value='15',allow_focus=True)
    input.set_click_function(input.set_on_focus)
    return input

#
# Functions when colors of cells clicked
#
def change_dead_cell_color(self):
    input = self.graphical_sprites.sprites()[SideBar.DEAD_COLOR_INPUT]
    index_color_actual = COLORS_LIST.index(input.background_color)
    new_color = COLORS_LIST[index_color_actual+1 if index_color_actual < len(COLORS_LIST)-1 else 0]
    grph.Graphics.cells_colors[0] = new_color
    input.background_color = new_color
    grph.Graphics.graphics.graphical_cells.update()
    grph.Graphics.graphics.internal_draw_elements.append(input); grph.Graphics.graphics.internal_draw = True
    grph.Graphics.graphics.updatable_elements.add(input);
        grph.Graphics.graphics.updatable_rects.append(input.rect)
    grph.Graphics.graphics.flip_screen = True

def change_alive_cell_color(self):
    input = self.graphical_sprites.sprites()[SideBar.ALIVE_COLOR_INPUT]
    index_color_actual = COLORS_LIST.index(input.background_color)
    new_color = COLORS_LIST[index_color_actual+1 if index_color_actual < len(COLORS_LIST)-1 else 0]
    grph.Graphics.cells_colors[1] = new_color
    input.background_color = new_color
    grph.Graphics.graphics.graphical_cells.update()
    grph.Graphics.graphics.internal_draw_elements.append(input); grph.Graphics.graphics.internal_draw = True
    grph.Graphics.graphics.updatable_elements.add(input);
        grph.Graphics.graphics.updatable_rects.append(input.rect)
    grph.Graphics.graphics.flip_screen = True
#
# Methods for actions in graphic elements and drawing
#
def move_drag_button(self,x):
    self.slider.move_drag_button(x)

def slider_move(self):
    SideBar.side_bar.move_drag_button(pygame.mouse.get_pos()[0])
    # cellular_automaton.update_zeros_density(SideBar.side_bar.slider.value)

def stop_button_pressed(self,button):
    self.graphical_sprites.sprites()[button].pressed_once = False

def draw(self,window):
    pygame.draw.rect(window,self.color_background,self.rectangle)
    # All the sprite elements get draw
    for element in self.graphical_elements:
        element.draw(window)
    self.graphical_sprites.draw(window)
    # Input text value
    self.graphical_sprites.sprites()[SideBar.RULE_INPUT].draw(window)
    # Inputs cells colors
    self.graphical_sprites.sprites()[SideBar.DEAD_COLOR_INPUT].draw(window)
    self.graphical_sprites.sprites()[SideBar.ALIVE_COLOR_INPUT].draw(window)

```

5.1.3 GraphicalComponents

Módulo relacionado a los gráficos, integrando de hecho clases que definen componentes gráficos con representación visual.

Control Definida como una clase modelo o abstracta, define el formato o esqueleto básico de muchos de los componentes *Sprite* en el mismo módulo.

MousePointer Única clase de componente gráfico sin representación visual en este módulo. Su principal objetivo es permitir identificar el colapso de otros componentes *Sprite* dentro de la interfaz con el cursor.

Consiste primordialmente en un *Sprite* rectángulo de una tamaño generalmente pequeño como de 1px por 1px, que se mueve a la misma posición que sigue el cursor.

CircularButton Clase que representa un componente gráfico con visualización, tiene generalmente un forma redondeada de botón. Esta trata de imitar la estética y funcionalidad de un botón con funciones al desencadenarse un evento(hover, click, stop_click, etc.)

Text Clase de componente gráfico que incluye texto en la interfaz gráfica. Esta clase provee métodos para el manejo sencillo del texto

Slider Clase de componentes gráfico compuesto con representación visual. El principal objetivo de esta clase es modelar la estética y funcionamiento normal de un *Slider* encontrado en cualquier interfaz grafica, proveyendo métodos para el manejo de eventos, etc.

Input Clase de componentes gráfico compuesto con representación visual. El principal objetivo de esta clase es modelar la estética y funcionamiento normal de un *Input* encontrado en cualquier interfaz grafica y más comunmente en formularios, proveyendo métodos para el manejo de eventos, etc.

Section Clase de componente gráfico con representación visual. Este componente funge únicamente como separador visual de secciones, y aunque podría utilizarse como referencia para la colocación de elementos, su objetivo principal es visual y nadamás.

```
import pygame
# User modules
from Constants import * # Mainly for the tuples of colors

class Control(pygame.sprite.Sprite):
    """Abstract class that defines the structure of a Control sprite graphical
    element like buttons
    """
    def __init__(self):
        super().__init__()
        self.pressed_once = False

    def update(self):
        pass

    def hover(self):
        pass

    def exit(self):
        pass

    def click(self):
        pass

    def stop_click(self):
        pass

    def has_on_focus(self):
        return False
```

```

class MousePointer(pygame.sprite.Sprite):
    """Simple class that provides an invisible rect for the pointer
    Used for easily identify collisions with the mouse pointer

Methods
-----
update(self) :
    Updates the position of the rect in the coodenates of the mouse
"""

def __init__(self,size:tuple):
    super().__init__()

    self.image = pygame.Surface(size)
    self.rect = self.image.get_rect()

def update(self):
    self.rect.x, self.rect.y = pygame.mouse.get_pos()

class Text():

    def __init__(self,position,text:str='Some Text',text_color:tuple=WHITE,font:pygame.font.Font=FONT):
        self.text = text
        self.font = font
        self.position = position
        self.text_color = text_color
        self.render = self.font.render(self.text, True, text_color)
        self.rect = pygame.Rect(self.position,self.render.get_rect().size)

    def update(self,new_text):
        self.text = new_text
        self.render = self.font.render(self.text, True, self.text_color)

    def print(self,window):
        window.blit(self.render,self.position)

class CircularButton(Control):

    def __init__(self,x,y,radius:int,image_size:int,image_path:str='',border:bool=True,background:tuple=DARK_BLACK,hover_background:tuple=LIGHT_GRAY,click_background:tuple=BLACK,click_function=None):
        super().__init__()
        self.id = id
        self.actual_background = background
        self.button_background = background
        self.button_hover_background = hover_background
        self.button_click_background = click_background

        self.border = border
        self.radius = radius
        self.image_size = image_size
        self.click_function = None

        # Controls that only 1 time the button gets pressed
        self.pressed_once = False
        self.ignore_multiple_click = False

        # The main surface gets created
        self.image = pygame.Surface((radius*2,radius*2))
        self.image.fill(DARK_BLACK)
        self.rect = self.image.get_rect(x=x,y=y,width=radius*2,height=radius*2)

        # Icon of button
        self.icon = None
        if image_path:

```

```

        self.icon = pygame.image.load(image_path)
        self.icon_rect = self.icon.get_rect(x=x,y=y,width=image_size,height=image_size)
        self.icon = pygame.transform.scale(self.icon,(image_size,image_size))

    # Circle of the outline
    if border: pygame.draw.circle(self.image,WHITE,(self.radius,self.radius), self.radius,width=1)

def update(self):
    # Background and icon
    pygame.draw.circle(self.image,self.actual_background,(self.radius,self.radius), self.radius-1 if
                      self.border else self.radius)
    if self.icon != None:
        self.image.blit(self.icon,(self.radius-self.image_size/2,self.radius-self.image_size/2))

def hover(self):
    self.actual_background = self.button_hover_background
    self.update()

def exit(self):
    self.actual_background = self.button_background
    self.update()

def click(self):
    if not self.pressed_once:
        self.actual_background = self.button_click_background
        self.click_function()
        self.update()
        if not self.ignore_multiple_click: self.pressed_once = True

def stop_click(self):
    self.pressed_once = False

def set_ignore_multiple_click(self):
    self.ignore_multiple_click = True

class Input(Control):

    def __init__(self,x,y,height,width:int=200,padding:int=5,label:str='Label',value:str='',allow_focus=False,background_color:DARK_GRAY):
        super().__init__()
        # The main surface gets created
        self.image = pygame.Surface((width,height))
        self.image.fill(DARK_BLACK)
        self.rect = self.image.get_rect(x=x,y=y,width=width,height=height)

        self.on_focus = False
        self.allow_focus = allow_focus

        self.clicked_function = lambda:print('clicked')
        self.new_character_function = None
        self.already_clicked = False

        # Label
        self.label_text = label
        self.label = Text((0,(height-FONT.size(label)[1])/2),label)
        self.label.print(self.image)
        # Background rectangle
        self.background_color = background_color
        self.background_rect_rect =
            pygame.Rect(FONT.size(label)[0]+2*padding,padding,width-FONT.size(label)[0]-2*padding,height-2*padding)
        # Input text
        self.value = value
        self.value_text = Text((self.rect.x+self.background_rect_rect.x+padding,
                               self.rect.y+padding),self.value)

```

```

#
# Control methods
#
def draw(self,window):
    pygame.draw.rect(self.image,self.background_color,self.background_rect_rect)
    pygame.draw.rect(self.image,BLUE if self.on_focus else LIGHT_BLACK_1 ,self.background_rect_rect,1)
    self.value_text.print(window)

def click(self):
    if not self.already_clicked:
        self.clicked_function()
    self.already_clicked = True

def stop_click(self):
    self.already_clicked = False

def has_on_focus(self):
    return self.allow_focus

#
# Auxiliar functions
#
def set_on_focus(self):
    self.on_focus = True

#
# External configuration methods
#
def new_character(self,character):
    if character == -1:
        if len(self.value) : self.value = self.value[:-1]
    else: self.value += character
    self.value_text.update(self.value)
    # Executes a function if specified when a new character gets processed into the value text
    if self.new_character_function != None: self.new_character_function()

def set_click_function(self,function):
    self.clicked_function = function


class Slider():

    def __init__(self,x:int,y:int,width:int,min:int=0,max:int=1):
        super().__init__()
        self.slider_background = DARK_BLACK
        self.corner_radius = 5
        self.width = width

        # The main surface gets created
        self.slider_surface = pygame.Surface((width,self.corner_radius*4))
        self.slider_surface.fill(DARK_BLACK)
        self.rect = self.slider_surface.get_rect(x=x,y=y,width=width,height=self.corner_radius*4)

        # Values of the slider between the limits
        self.value = 0.5

        # Coordenates values
        self.y_center = self.corner_radius*2
        self.reference_rectangle_width = self.width/2 - self.corner_radius

        # Drag circular button
        self.drag_button_group = pygame.sprite.Group()
        self.drag_button = CircularButton(self.rect.x+(self.rect.width/2)-self.corner_radius*2, self.rect.y,
                                         self.corner_radius*2, 0, background=WHITE, hover_background=WHITE, click_background=GRAY,
                                         border=1, border_color=WHITE, font_size=16, font_color=BLACK, text='0')

```

```

        border=False, id='Drag Button')
self.drag_button.set_ignore_multiple_click() # Allow to drag the button while keep pressing the click
self.drag_button_group.add(self.drag_button)

# Base geometric shapes
pygame.draw.circle(self.slider_surface,WHITE,( self.corner_radius,      self.y_center),
                   self.corner_radius)
pygame.draw.circle(self.slider_surface,WHITE,( self.width-self.corner_radius, self.y_center),
                   self.corner_radius,width=1)
pygame.draw.rect(self.slider_surface,WHITE,[ self.corner_radius,
                                             self.y_center-self.corner_radius, self.rect.width-self.corner_radius*2, self.corner_radius*2
                                             ],width=1)
# Circle to hide the intersection between the right circle corner and the rect
pygame.draw.rect(self.slider_surface,DARK_BLACK,[self.width-self.corner_radius*2,
                                                 self.corner_radius*1.5-1, self.corner_radius*1.5, (self.corner_radius*2)-2])
# Texts
self.value_text = Text((self.rect.x+(self.width/2)-(FONT.size('0.500')[0]/2),
                       self.rect.y+(self.corner_radius*4)+10),"0.500")
self.texts = [
    Text((self.rect.x+5, self.rect.y+(self.corner_radius*4)+8.5),"0's",font=SMALL_FONT),
    Text((self.rect.x+self.width-self.corner_radius-15,
          self.rect.y+(self.corner_radius*4)+8.5),"1's",font=SMALL_FONT),
    self.value_text
]
]

# Defines the limits of the drag button
self.min_limit = self.rect.x
self.max_limit = self.rect.x+self.rect.width - self.corner_radius*2

def draw(self,window):
    for text in self.texts:
        text.print(window)

    window.blit(self.slider_surface,self.rect)
    pygame.draw.rect(window,WHITE,[self.rect.x +
                                   self.corner_radius,self.rect.y+self.corner_radius,self.reference_rectangle_width,self.corner_radius*2])

def update_value(self):
    if self.reference_rectangle_width > 0.0:
        self.value = self.reference_rectangle_width/(self.width-self.corner_radius*2)
    else: self.value = 0.0
    self.value_text.update('{:.3f}'.format(self.value))

def get_drag_button(self):
    return self.drag_button

def move_drag_button(self,x):
    x -= self.corner_radius*2
    if (x <= self.max_limit) and (x >= self.min_limit):
        self.reference_rectangle_width = x - self.rect.x
        self.drag_button.rect.x = x
        self.update_value()

class Section():

    def __init__(self,x,y,width,padding,title):
        self.header_surface = pygame.Surface((width,30))
        self.header_surface.fill(LIGHT_BLACK_1)
        self.header_rect = self.header_surface.get_rect(x=x,y=y,width=width,height=30)

        # The title of the section
        self.title_text = Text((padding,9),title,font=SMALL_FONT)
        self.title_text.print(self.header_surface)

```

```
def draw(self,window):
    window.blit(self.header_surface,self.header_rect)
```

5.1.4 Graphics

Módulo que integra en su definición la clase **Graphics**, la cuál bien podría tratarse como un nodo del sistema, encargándose en específico de la coordinación gráfica entre elementos gráficos, eventos y comunicación con el nodo lógico principal de la simulación. Su comportamiento definido por los métodos y atributos que maneja podría definirla como una clase *Front-End* del programa de simulación.

Así mismo conjunta una segunda clase llamada **Cell**, siendo la definición de una célula gráfica. Esta clase hereda de la clase de PyGame *Sprite*, permitiéndole verificar colisiones con otros *Sprites*, sin embargo no solo es una definición de esta clase gráfica, pues se añaden algunos métodos más que le permiten modificar su comportamiento y valores, exponiendo sus métodos como principal interfaz para la interacción con instancias de esta clase.

```
from queue import Queue
import queue
from matplotlib.pyplot import plot
import pygame
import numpy as np
# User modules
import time
import main as mn
from ECA import ECA,Attractors
from Constants import *
from Layouts import Grid,BottomBar,SideBar
from GraphicalComponents import Control,MousePointer


class Graphics():
    """Class in charge of the interface aspects of the program
    Implemented as a Singleton class as just 1 instance will be
    necessary to control the graphical of the program

    Functions
    -----
    get_graphics( grid, side_bar, bottom_bar ) : graphics
        Returns the Singleton instance of the class
    """
    #=====
    # Class properties
    #=====
    graphics = None
    cells_colors = [WHITE,LIGHT_BLACK_2]

    #=====
    # Class methods
    #=====
    def set_cells_colors(colors):
        Graphics.cells_colors = colors

    def set_alive_color(color):
        Graphics.cells_colors[1] = color

    def set_dead_color(color):
        Graphics.cells_colors[0] = color

    def __init__(self, grid:Grid, side_bar:SideBar=None, bottom_bar:BottomBar=None):
        Graphics.graphics = self

        # Interface drawable sprite elements
        # self.drawable_elements = []
```

```

# Grid for the cells
self.grid = grid

# Array of graphical cells
self.graphical_cells = pygame.sprite.Group()
self.updateable_rects = [] # List of individual rects of each cells that must be updated
self.updateable_elements = pygame.sprite.Group() # Group of sprites elements/cells that must be re-draw
# before update
self.create_graphical_cells()

# Logical ECA
self.eca = ECA(self.grid.num_cols,self.grid.num_rows,self.graphical_cells.sprites())
self.delay_generations = time.time()

# Collapsable pointer
self.pointer = MousePointer((1,1))

# Graphical elements and sections
self.side_bar = side_bar
self.bottom_bar = bottom_bar
# Graphical elements that also can be interacted with
self.collidable_elements = pygame.sprite.Group()
self.collidable_elements.add(self.side_bar.get_graphical_sprites())

#
# Global status variables
#
self.play = False
self.done = False
self.flip_screen = False
self.click_pressed = False
self.internal_draw = False
self.internal_draw_elements = []
self.update_evolution_parameters = False # Implies the update of the number of generation and number of
# alive cells
# self.last_digit_typed = ''
# Used for changing the states of collaideable elements

# List of functions to be executed in an independent thread just for this kind of functions
# this way, nor the graphical or events functions, gets delayed until this finishes
self.execute_functions = []

self.assign_functions_to_clicked_buttons()

=====
# Setters and getters
=====

def set_delay_generations(self,delay):
    self.delay_generations = delay

=====

# Graphical functions
=====

def create_graphical_cells(self):
    cell_size = self.grid.calculate_element_sizes()
    for rows in range(self.grid.num_rows):
        for columns in range(self.grid.num_cols):
            self.graphical_cells.add(Cell(cell_size,(columns,rows)))

def assign_functions_to_clicked_buttons(self):
    self.side_bar.set_click_function(SideBar.PLAY_BUTTON,self.Play)
    self.side_bar.set_click_function(SideBar.STOP_BUTTON,self.stop)
    self.side_bar.set_click_function(SideBar.RESTART_BUTTON,self.restart)

```

```

self.side_bar.set_click_function(SideBar.INITIAL_CENTER_CELL_BUTTON, self.initial_center_cell)
self.side_bar.set_click_function(SideBar.RANDOM_INITIAL_BUTTON, self.initial_random_configuration)
self.side_bar.set_click_function(SideBar.DENSITY_BUTTON, self.add_plot_density_functions)
self.side_bar.set_click_function(SideBar.LOGARITHM_BUTTON, self.add_plot_density_logarithm_functions)
self.side_bar.set_click_function(SideBar.ENTROPY_BUTTON, self.add_plot_shannons_entropy_functions)
self.side_bar.set_click_function(SideBar.CLEAR_BUTTON, self.clean)
self.side_bar.set_click_function(SideBar.UPLOAD_BUTTON, self.upload_evolution_space)
self.side_bar.set_click_function(SideBar.SAVE_BUTTON, self.save_evolution_space)
self.side_bar.set_click_function(SideBar.ATTRACTORS_BUTTON, self.compute_attractors)

=====
# Pygame Loop functions
=====

def look_for_events(self):
    clock = pygame.time.Clock()
    while not self.done:
        clock.tick(45)
        # Look for every possible event in the programm
        for event in pygame.event.get():
            # When pressing close button in top bar
            if event.type == pygame.QUIT:
                self.done = True
                return True

            # When mouse button pressed
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1: # Left click
                    self.click_pressed = True
            elif event.type == pygame.MOUSEBUTTONUP:
                if event.button == 1: # Left click
                    self.click_pressed = False
            # Looks for mouse wheel events
            elif event.type == pygame.MOUSEWHEEL:
                # Scrolling up
                if event.y == 1:
                    ECA.eca.scroll_space(up=True)
                # Scrolling down
                elif event.y == -1:
                    ECA.eca.scroll_space(up=False)

        """
        Key map
        ++++++
        Keys use for execute actions in the programm

        p : Plays the continuous evolution process
        n : Computes only 1 generation of the space evolution
        s : Stops the evolution process
        c : Cleans the evolutions grids and restart the dynamic variables for the evolution
            process
        m : Sets the central cell of an evolution space in the actual row as alive

        """

        # Looks for key press
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_p: self.Play()
            elif event.key == pygame.K_n: self.execute_functions.append(self.next_generation);
                self.one_execution = True; print('<-- Step next generation -->')
            elif event.key == pygame.K_s: self.stop()
            elif event.key == pygame.K_c: self.execute_functions.append(self.clean)
            elif event.key == pygame.K_m: self.execute_functions.append(self.initial_center_cell)
            elif event.key == pygame.K_BACKSPACE: self.last_digit_typed = -1
            else:
                aux_digit = event.unicode

```

```

        if ord(aux_digit) >= 48 and ord(aux_digit) <= 57:
            self.last_digit_typed = event.unicode
        else: print('<!!! Only digits are allowed !!!>')

def events_actions(self):
    clicked_elements = set(); hovered_elements = set(); focused_element = []
    last_generation_time = time.time()
    clock = pygame.time.Clock()
    while not self.done:
        clock.tick(45)
        # Updates the collidable pointer position
        self.pointer.update()

        # Verifies if a collapsible element of the UI collided
        collaided_elements = pygame.sprite.spritecollide(self.pointer,self.collidable_elements,False)
        # Verifies if a cell has colapsed when there's been a click
        if self.click_pressed: collaided_elements +=
            pygame.sprite.spritecollide(self.pointer,self.graphical_cells,False)

        # Looks for previously focused elements so they can change the focus state
        if focused_element:
            # As the only focused element in the programm
            # when a new character gets typed is changed
            # in the input text showed
            if self.last_digit_typed != '':
                focused_element[0].new_character(self.last_digit_typed)
                # Only the input of the rule gets to update the actual rule
                if focused_element[0].label_text == 'Rule':
                    self.eea.update_rule(int(focused_element[0].value) if focused_element[0].value != '' else
                                         0)
                # Adds the graphical element to be draw and updated
                self.updatable_elements.add(element)
                self.internal_draw = True; self.internal_draw_elements.append(element)
                self.updatable_rects.append(element.rect)
                self.last_digit_typed = ''
            if self.click_pressed:
                if focused_element[0] not in collaided_elements:
                    element = focused_element.pop(); element.on_focus = False
                    # Adds the graphical element to be draw and updated
                    self.updatable_elements.add(element)
                    self.internal_draw = True; self.internal_draw_elements.append(element)
                    self.updatable_rects.append(element.rect)

        # Looks for previously clicked elements/cells so they can stop_click()
        if clicked_elements and not self.click_pressed: # Stop clicking
            for clicked in clicked_elements:
                clicked.stop_click()
            # Adds the graphical element to be draw and updated
            self.updatable_elements.add(clicked)
            self.updatable_rects.append(clicked.rect)
            clicked_elements.clear()

        # Looks for previously hovered elements so they can exit()
        if hovered_elements:
            remove_hovered = []
            for hovered in hovered_elements:
                # Identifies if the element no longer collides with the pointer
                if hovered not in collaided_elements:
                    hovered.exit()
                    remove_hovered.append(hovered)
                # Adds the graphical element to be draw and updated
                self.updatable_elements.add(hovered)
                self.updatable_rects.append(hovered.rect)
            # The elements get removed from the set
            if remove_hovered :

```

```

        for removed in remove_hovered: hovered_elements.remove(removed)

# At least an element has collaided
if collaided_elements:
    for element in collaided_elements:
        if self.click_pressed: # It's been clicked
            # It's a graphical cell
            if hasattr(element,'index'):
                # Only can be modified graphical cells of the actual generation row
                if element.index[1] == self.eca.actual_row:
                    element.click()
                    clicked_elements.add(element)
                    # Adds the cell to be draw and updated
                    self.updatable_elements.add(element)
                    self.updatable_rects.append(element.rect)# At least an element has collaided
            # Other graphical elements like buttons
        else:
            # Gets the click function of the element to be executed in the thread of functions
            element.click()
            clicked_elements.add(element)
            # Graphical elements with focus
            if element.has_on_focus():
                focused_element.insert(0,element)
                self.last_digit_typed = ''
                # Adds the graphical element to be draw and updated
                self.updatable_elements.add(element)
                self.internal_draw = True; self.internal_draw_elements.append(element)
                self.updatable_rects.append(element.rect)
            if not self.click_pressed and not element.has_on_focus(): # There's no click pressed
                element.hover()
                hovered_elements.add(element)
                # Adds the graphical element to be draw and updated
                self.updatable_elements.add(element)
                self.updatable_rects.append(element.rect)

# Verifies the time elapsed since the last generation been computed
if self.play:
    if time.time()-last_generation_time >= self.delay_generations:
        self.execute_functions.append(self.next_generation)
        last_generation_time = time.time()

def draw_elements(self,window):
    if self.update_evolution_parameters:
        # Update of the evolution variables
        self.updatable_rects.append(self.bottom_bar.update_alive_cells(self.eca.alive_cells))
        self.updatable_rects.append(self.bottom_bar.update_generations(self.eca.generations))
        self.bottom_bar.draw(window)
        self.update_evolution_parameters = False
    # Only draws the elements when the display has to be flipped or when there's updatable rects
    if self.flip_screen or self.updatable_rects:
        # Draws and updates all the graphical cells
        if self.flip_screen:
            self.bottom_bar.draw(window)
            self.side_bar.draw(window)
            self.graphical_cells.draw(window)
            pygame.display.flip()
            self.flip_screen = False
        # Draws and updates only the cells that changed of color
    else:
        aux_updatable_rects = (list(self.updatable_rects)); del self.updatable_rects[:]
        self.updatable_elements.draw(window); self.updatable_elements.empty()
        # There's elements that has a defined draw function needed to be executed so that all the
        # graphical elements get shown
        if self.internal_draw:
            while self.internal_draw_elements: self.internal_draw_elements.pop(0).draw(window)

```

```

        self.internal_draw = False
        pygame.display.update(aux_updatable_rects)

#=====
#     Action functions
#=====

def execution_of_functions(self,queue_main:Queue):
    clock = pygame.time.Clock()
    while not self.done:
        clock.tick(30)
        if self.execute_functions:
            (self.execute_functions.pop(0))(share=queue_main)

def next_generation(self,**kargs):
    if self.play or self.one_execution:
        aux_rects,aux_cells,proceed = self.eca.compute_next_generation()
        # Statistical analysis
        self.eca.density()
        self.eca.density_logarithm()
        self.eca.shannon_entropy()
        if proceed:
            self.update_evolution_parameters = True
            # Both the rects and sprites of cells gets added to the respective global variable
            self.updatable_rects += aux_rects; self.updatable_elements.add(aux_cells)
        else:
            self.play = False
            print('<--- Evolution process stoped for the space is to small to keep evolving --->')
            self.one_execution = False

def upload_evolution_space(self,**kargs):
    self.clean()
    aux_updatable_rect,aux_updatable_cells = self.eca.upload_evolution_space()
    self.update_evolution_parameters = True
    self.updatable_elements.add(aux_updatable_cells); self.updatable_rects += aux_updatable_rect

def save_evolution_space(self,**kargs):
    self.eca.save_evolution_space()

def Play(self,**kargs):
    self.play = True
    print('<--- Running evolution process --->')

def stop(self,**kargs):
    self.play = False
    print('<--- Pausing evolution process --->')

def initial_center_cell(self,**kargs):
    self.eca.initial_center_cell()
    self.flip_screen = True

def initial_random_configuration(self,**kargs):
    self.eca.initial_random_configuration()
    self.flip_screen = True

def clean(self,**kargs):
    self.eca.clean()
    self.flip_screen = True
    print('<--- Cleaning of space --->')

def restart(self,**kargs):
    self.clean()
    # Calls the logical ECA restart
    aux_updatable_rect,aux_updatable_cells = self.eca.restart()
    self.updatable_elements.add(aux_updatable_cells); self.updatable_rects += aux_updatable_rect

```

```

def compute_attractors(self,**kargs):
    self.clean()
    Attractors.attractors.compute_attractors()

#
# Statistical analysis
#
# Function that adds the plotting functions into the function execution thread so that
# it can share the matplotlib function to the main thread through the queue object
def add_plot_density_functions(self): self.execute_functions.append(self.plot_density)
def add_plot_density_logarithm_functions(self): self.execute_functions.append(self.plot_density_logarithm)
def add_plot_shannons_entropy_functions(self): self.execute_functions.append(self.plot_shannons_entropy)

def plot_density(self,share:Queue=None):
    share.put(self.eca.plot_density)

def plot_density_logarithm(self,share:Queue=None):
    share.put(self.eca.plot_density_logarithm)

def plot_shannons_entropy(self,share:Queue=None):
    share.put(self.eca.plot_shannons_entropy)


class Cell(Control):

    def __init__(self, size:tuple, index:tuple, alive:bool=False):
        super().__init__()
        # Status variable for when is clicked once
        self.already_clicked = False
        # Cells properties
        self.alive = alive
        # Index of the cell in the general grid
        self.index = index
        # The cell and the rect inside gets defined
        self.image = pygame.Surface(size)
        self.image.fill(Graphics.cells_colors[int(self.alive)])
        # The rect of the cell is assigned so in
        # further methods can be used to move it
        self.rect = self.image.get_rect()

    -----
    # Functions of the Control class
    -----
    def update(self):
        # Updates the background color of the cell
        self.image.fill(Graphics.cells_colors[int(self.alive)])

    def exit(self):
        self.already_clicked = False

    def click(self):
        # Changes the alive status of the cell and reprints it's color
        if not self.already_clicked:
            self.already_clicked = True
            self.alive = not self.alive
            # Updates the logical cells
            ECA.eca.update_cell_status(self.index[0])
            self.update()

    def stop_click(self):
        self.already_clicked = False

    -----

```

```

# Cell methods
-----
def move(self,coord:tuple):
    self.rect.x = coord[0]
    self.rect.y = coord[1]

def set_status(self,status):
    self.alive = status
    self.update()

```

5.1.5 ECA

ECA Clase que por definición de las responsabilidades, métodos y atributos que maneja, puede ser considerado uno de los Nodos principales de la simulación. Esta clase se encarga del manejo de la lógica detrás de la simulación, definiéndose en esta métodos encargados de la configuración, evolución y comunicación externa para los espacios de evolución. Evidentemente y al igual que sucede con la clase **Graphics**, tiene una relación directa de comunicación con el nodo principal gráfico para mantener la consistencia de los espacios de células lógicos y gráficos.

Los métodos de esta clase principalmente se enfocan en la creación, configuración, reinicio y evolución del espacio de autómatas, integrando a su vez métodos auxiliares que se exponen como interfaces de comunicación con el nodo gráfico de la simulación. Se cuenta también con métodos encargados de la graficación de las diferentes mediadas estadísticas de los espacios, y el guardado, alzado de espacios desde archivos CSV y el scrolling.

Attractors Clase lógica que se enfoca en el cálculo completo de los atractores en un rango de potencias para la regla específica.

Además del cálculo de los atractores, se tienen funciones para el cálculo de las cadenas que conforman el universo potencia binario y la graficación de los nodos identificados en cada evolución y que son guardados en la carpeta *graphs* ya como una imagen con extensión *eps*.

```

import time
import random
import numpy as np
import igraph as ig
import networkx as nx
from math import log10,log2
import matplotlib.pyplot as plt
# Users module
from Constants import COLOUR_ATTRACTOR, MATRIX_BIN_TO_DEC,BITS_MASKS,HEX_COLORS_LIST
from Layouts import SideBar as SB
import Graphics as Grph
import Layouts as Ly

class ECA():

    # Singleton Class
    eca = None

    def __init__(self,number_cells_1D, evolutionary_space_rows,graphical_cells):
        ECA.eca = self
        self.attractors = Attractors(self)
        # Defines the initial size of the evolving and saving spaces
        # Adds 2 columns in cells for the toroidal behavior
        self.number_cells_1D = number_cells_1D
        self.dimensions = (evolutionary_space_rows,self.number_cells_1D+2)
        # Grid of 2 dimensions for saving the evolutions in time of the ECA in 1D
        self.saving_space = np.zeros(self.dimensions,np.ubyte)
        # Defines an evolution space in which the evolutions will be computed
        # It has 2 rows, the first for the actual space combination and the second
        # for the resultant evolution
        self.evolution_space = np.zeros((2,self.dimensions[1]),np.ubyte)
        # Reference in list of the graphical cells
        self.graphical_cells = graphical_cells
        # States
        self.zeros_density = 0.5

```

```

# Dynamic variables through the process
self.generations = 0
self.alive_cells = 0
# Defines the actual row that is actually getting evolved
self.actual_row = 0
# Rule number
self.rule = 110
# Identifies the saved spaces and the showed spaces. Used for scrolling
self.evolution_spaces_saved = 0
self.actual_evolution_space = 1
# Record of statistical analysis
self.density_record = []
self.density_logarithm_record = []
self.shannon_entropy_record = []

#-----
# Configuration and button pressed actioned methods
#-----

def initial_random_configuration(self):
    updatable_rects = []; updatable_cells = []
    for i in range(self.number_cells_1D):
        if self.zeros_density > random.random(): # Probability of an alive cell is the complement of the
            state 0
        # Updates the alive cell in all the evolutions spaces and graphical cells
        self.saving_space[0,i+1] = 1
        self.evolution_space[0,i+1] = 1
        self.graphical_cells[i].alive = True
        self.alive_cells += 1
        # Updates the interface
        self.graphical_cells[i].update()
        updatable_cells.append(self.graphical_cells[i])
        updatable_rects.append(self.graphical_cells[i].rect)
    return updatable_rects,updatable_cells

def initial_center_cell(self):
    center_index = int(self.number_cells_1D/2)
    # Updates the alive cell in all the evolutions spaces and graphical cells
    self.saving_space[0,center_index+1] = 1
    self.evolution_space[0,center_index+1] = 1
    # Graphical cell
    self.graphical_cells[center_index].alive = True
    self.graphical_cells[center_index].update()
    self.alive_cells += 1
    print('<-- Center cell (x:{})) --->'.format(center_index))

def expand_evolution_space(self,length_tail:int=5):
    self.evolution_spaces_saved += 1
    self.save_evolution_space(True, self.evolution_spaces_saved)
    self.actual_evolution_space += 1
    # A tail rows of the las evolution space gets copied to the head of the new evolution space
    self.saving_space[0:length_tail,:] = self.saving_space[-length_tail:self.dimensions[0],:]
    # The saving space gets cleared
    self.saving_space[length_tail:,:] = 0
    # Update of the actual row
    self.actual_row = length_tail-1

    # Graphical cells
    for y in range(self.dimensions[0]):
        for x in range(1,self.number_cells_1D+1):
            aux_cell = self.graphical_cells[y*self.number_cells_1D+(x-1)]
            # Verifies the values of the graphical cells of the rows of the heads space
            if y < length_tail:
                if not(aux_cell.alive == bool(self.saving_space[y,x])):
                    aux_cell.alive = not(aux_cell.alive)
            else: aux_cell.alive = False

```

```

        aux_cell.update()

Ly.BottomBar.bottom_bar.update_dimensions_number_space([ECA.eca.dimensions[0],ECA.eca.number_cells_1D],self.actu
Grph.Graphics.graphics.flip_screen = True

def clean(self,also_spaces_saved:bool=True):
    # The evolutions spaces gets all the cells restart to dead
    self.saving_space[self.actual_row,:] = 0
    self.evolution_space[0,:] = 0
    for i in range(self.number_cells_1D*(self.actual_row+1)):
        self.graphical_cells[i].alive = False
        self.graphical_cells[i].update()
    # Dinamyc changing variables
    self.actual_row = 0
    self.alive_cells = 0
    self.generations = 0
    if also_spaces_saved:
        self.evolution_spaces_saved = 0
        self.actual_evolution_space = 1
    # Statistical records
    self.density_record.clear()
    self.density_logarithm_record.clear()
    self.shannon_entropy_record.clear()
    # Graphical texts in the bottom
    Ly.BottomBar.bottom_bar.update_dimensions_number_space([ECA.eca.dimensions[0],ECA.eca.number_cells_1D],self.actu
    Ly.BottomBar.bottom_bar.update_alive_cells(0)
    Ly.BottomBar.bottom_bar.update_generations(0)

def restart(self):
    self.clean()
    return self.initial_random_configuration()

def update_zeros_density(self,density):
    self.zeros_density = density

def update_rule(self,rule):
    self.rule = rule

def update_cell_status(self,cell_position, status:bool=True, invert:bool=True,graphical_cell:bool=False):
    # When invert active changes the state of the cell to the contrary of the current cell state
    status = (not self.evolution_space[0,cell_position+1]) if invert else status
    self.evolution_space[0,cell_position+1] = int(status)
    self.saving_space[self.actual_row,cell_position+1] = self.evolution_space[0,cell_position+1]
    # Also updates the graphical cells
    if graphical_cell:
        self.graphical_cells[self.actual_row*self.number_cells_1D + cell_position].alive = status
        self.graphical_cells[self.actual_row*self.number_cells_1D + cell_position].update()
    return self.graphical_cells[self.actual_row*self.number_cells_1D + cell_position],
           self.graphical_cells[self.actual_row*self.number_cells_1D + cell_position].rect

def save_evolution_space(self,internal_space:bool=False,id_internal_saved:int=1):
    if not internal_space: filename = './saves/Rule_{0}_generation_{1}.csv'.format(self.rule,self.generations)
    # Used when saving an space to allow more evolutions by clearing rows
    else: filename = './internal_spaces/{0}.csv'.format(id_internal_saved)
    np.savetxt(
        filename,
        self.saving_space[:self.actual_row+1,1:-1],
        delimiter = ',',
        fmt = '% s'
    )
    if not internal_space: print('<-- File successfully saved as \'{}\' -->'.format(filename))

def upload_evolution_space(self,internal_space:bool=False,id_internal_saved:int=1):
    aux_array = np.genfromtxt("./saves/upload.csv" if not internal_space else

```

```

'./internal_spaces/{}.csv'.format(id_internal_saved), delimiter=', ', ); aux_shape = aux_array.shape
if (self.dimensions[0]-aux_shape[0] < 0) or (self.dimensions[1]-aux_shape[1] < 0):
    print('!!! The actual evolution space is smaller than the intended upload file !!!')
# The upload array gets loaded in the programm
else:
    updatable_rects = []; updatable_cells = []
    # First the space gets cleaned
    self.clean(also_spaces_saved=not internal_space)
    initial_column = int((self.dimensions[1]-aux_shape[1])/2)
    # The new array gets saved in the saving_space and the evolution_space arrays
    self.saving_space[:aux_shape[0],initial_column:initial_column+aux_shape[1]] = aux_array
    self.evolution_space[0,initial_column:initial_column+aux_shape[1]] = aux_array[-1,:] # The las row
        of the uploaded array gets copied to the evolution space
    # The graphical cells gets updated with the value of the new array
    for y in range(aux_shape[0]):
        for x in range(initial_column,initial_column+aux_shape[1]):
            if self.saving_space[y,x]:
                aux_cell = self.graphical_cells[y*self.number_cells_1D+(x-1)]
                aux_cell.alive = True
                aux_cell.update()
                updatable_cells.append(aux_cell)
                updatable_rects.append(aux_cell.rect)

    # Dynamic variables
    self.alive_cells = int(aux_array[-1].sum())
    self.generations = aux_shape[0]
    self.actual_row = self.generations-1
    if not internal_space: print('<--- New configuration successfully uploaded --->')
    return updatable_rects,updatable_cells

def scroll_space(self,up=True):
    # No evolution space before/after to be scroll
    if up and self.actual_evolution_space == 1:
        print('<!!! No previous evolutions exists to scroll upward !!!>')
        return False
    elif not up and self.evolution_spaces_saved <= self.actual_evolution_space:
        print('<!!! No more evolutions exists to scroll downward !!!>')
        return False

    if up:
        print('<--- Scrolling up --->')
        # It's the last evolution space
        if self.actual_evolution_space > self.evolution_spaces_saved:
            self.evolution_spaces_saved += 1
            self.save_evolution_space(internal_space=True,id_internal_saved=self.evolution_spaces_saved)
        self.actual_evolution_space -= 1
        # The previously saved space gets loaded
        self.upload_evolution_space(internal_space=True,id_internal_saved=self.actual_evolution_space)

    if not up:
        print('<--- Scrolling down --->')
        # When scrolling down to the last evolution space
        self.actual_evolution_space += 1
        self.upload_evolution_space(internal_space=True,id_internal_saved=self.actual_evolution_space)
        if self.evolution_spaces_saved == self.actual_evolution_space: self.evolution_spaces_saved -= 1

Ly.BottomBar.bottom_bar.update_dimensions_number_space([ECA.eca.dimensions[0],ECA.eca.number_cells_1D],self.actu
Grph.Graphics.graphics.flip_screen = True

#-----
# Auxiliar Methods
#-----
def toroidal_padding(self):
    self.evolution_space[0] = self.evolution_space[-2]
    self.evolution_space[-1] = self.evolution_space[1]

```

```

def get_actual_evolution_row(self,window:tuple=(1,-1)):
    return self.evolution_space[0,window[0]:window[1]]

def update_evolution_row(self,jump:int=1):
    self.actual_row += jump
    if self.actual_row >= self.dimensions[0]: self.expand_evolution_space()

#-----
# Evolution space related methods
#-----
def compute_next_generation(self,restrain_window=False,window_range=(0,-1)):
    # Verifies that the next evolution exceeds the actual size of the array
    if (self.actual_row == self.dimensions[0]-1):
        self.expand_evolution_space()

    window_range = (0,self.number_cells_1D) if not restrain_window else (window_range[0]-1,window_range[1])
    updatable_rects = []; updatable_cells = []
    self.toroidal_padding()
    self.alive_cells = 0
    for x in range(window_range[0],window_range[1]):
        window = np.copy(self.evolution_space[0,x:x+3])
        # Converts to decimal and sums the value to get the index of the bit mask
        index_neighbourhood = (window*MATRIX_BIN_TO_DEC).sum()
        # With the bit mask gets the value of the AND bit operation
        # when the value is greater than 0 the value of the new cell is 1
        new_cell_status = self.rule & BITS_MASKS[index_neighbourhood]
        if new_cell_status :
            new_cell_status = 1
            self.alive_cells += 1
        # The result gets loaded in the evolution space second row
        self.evolution_space[1,x+1] = new_cell_status
        # Only when the new_status cell changes to alive
        if new_cell_status:
            aux_cell = self.graphical_cells[(self.actual_row+1)*self.number_cells_1D+x]
            aux_cell.alive = True; aux_cell.update()
            updatable_rects.append(aux_cell.rect)
            updatable_cells.append(aux_cell)
        # Once the new states have been calculated, the new evolutions results get saved
        self.saving_space[self.actual_row+1,:] = self.evolution_space[1,:]
        self.evolution_space[0,:] = self.evolution_space[1,:]
        self.evolution_space[1,:] = 0 # Cleans the second row with 0's
        self.generations += 1
        self.actual_row += 1
        # Returns a list of the rects of the cells that changed value
        return list(updatable_rects),updatable_cells,True
#
# Statistical analysis
#
def density(self):
    self.density_record.append(int(self.alive_cells))

def density_logarithm(self):
    self.density_logarithm_record.append(log10(self.alive_cells))

def shannon_entropy(self):
    entropy = 0
    neighbourhood_frecuency = [ 0 for i in range(8) ]

    for x in range(self.number_cells_1D):
        window = np.copy(self.evolution_space[0,x:x+3])
        neighbourhood_number = (window*MATRIX_BIN_TO_DEC).sum()
        neighbourhood_frecuency[neighbourhood_number] += 1

    probability = 0.0

```

```

for frecuency in neighbourhood_frecuency:
    if frecuency: # Different of 0
        probability = frecuency/self.number_cells_1D
        entropy -= probability*log2(probability)

    self.shannon_entropy_record.append(entropy)

def plot_density(self):
    if not self.density_record:
        print('<!!! No density of cells has been recorded !!!>')
        return False

    fig = plt.figure()
    ax = plt.axes()

    ax.set( xlabel = 'Generations',
            ylabel = 'Density',
            title = 'Alive Cells Density'
        )
    ax.grid()

    data = list(self.density_record)
    data = np.array(list(enumerate(list(data))))
    ax.scatter(data[:,0],data[:,1])
    ax.plot(data[:,0],data[:,1])

    plt.show()

def plot_density_logarithm(self):
    if not self.density_logarithm_record:
        print('<!!! No logarithm density of cells has been recorded !!!>')
        return False

    fig = plt.figure()
    ax = plt.axes()

    ax.set( xlabel = 'Generations',
            ylabel = 'Density Logarithm',
            title = 'Logarithm Alive Cells Density'
        )
    ax.grid()

    data = list(self.density_logarithm_record)
    data = np.array(list(enumerate(list(data))))
    ax.scatter(data[:,0],data[:,1])
    ax.plot(data[:,0],data[:,1])

    plt.show()

def plot_shannons_entropy(self):
    if not self.shannon_entropy_record:
        print('<!!! No shannons entropy has been recorded !!!>')
        return False

    fig = plt.figure()
    ax = plt.axes()

    ax.set( xlabel = 'Generations',
            ylabel = 'Entropy',
            title = 'Shannons Entropy'
        )
    ax.grid()

    data = list(self.shannon_entropy_record)
    data = np.array(list(enumerate(list(data))))

```

```

ax.scatter(data[:,0],data[:,1])
ax.plot(data[:,0],data[:,1])

plt.show()

class Attractors():
    """
    Libraries
    -----
    igraph :
        Use to describe a graph defining their vertices and relations between them

    cairocffi :
        Allows igraph to plot the graph in a 2D image
    """
#-----
# Class's function and variable
#-----
attractors = None
def generate_binary_chains(power:int,previous_chains=[]) -> list:
    """Function that generates the binary chains of all the universe starting from the
    power 1 until the give power

    Parameters
    -----
    power : int
        Finish power of the chain universe to be calculated

    Returns
    -----
    chains : list
        Bidimensional list of the chains generated for each power universe.
        Every element in the second dimension holds the chains of a power universe
    """
    # When power is 1 or less only the binary universe of chains in power 1 is returned
    if power <= 1: return [ ["0","1"] ]
    # Validates if a given list with previously calculated chains has been given
    chains = previous_chains
    if not chains: chains = [ ["0","1"] ]
    # Loop of the consecutive universe of powers
    for p in range(len(chains)-1,power):
        chains.append([]); print('p >>',p)
        # Loop for the generation of the chains of a given universe power
        for previous_chain in chains[p]:
            chains[p+1].extend([previous_chain+str(0), previous_chain+str(1)])

    return chains

def __init__(self,eca):
    Attractors.attractors = self
    # Range of the sizes of evolutions spaces from which will be calculated its attractors
    self.sizes = (0,5)
    # List containing all the possible combinations of values of the cells in a space from a range of sizes
    self.binary_chains = []
    self.conversion_matrix = np.array([1])
    # Trees of the attractors found in each universe
    # The first dimension of the dictionary will be the trees of each universe and the key corresponds to
    # the number of power of the universe
    #     Inside each universe dictionary there's a dictionary with keys:
    #         relations : Relations between nodes
    self.trees = dict()

```

```

self.actual_universe = 0
self.list_attractors = list()
# When using matplotlib for networkx
""" self.sizes_graph = [
    (0,0),
    (4,4),
    (4,4),
    (4,4),
    (6,6),
    (10,10),
    (14,14),
    (34,34),
    (46,46),
    (60,60),
    (80,80),
    (115,115),
    (165,165),
]
"""
self.sizes_igraph = [
    0,
    100,
    140,
    200,
    270,
    350,
    450,
    750,
    1700,
    1700,
    1500,
    1700,
    2300,
    3200,
    4000,
    4900,
    5700,
    6800,
    6800,
    6800,
    6800,
]
]

#-----
# Auxiliar Methods
#-----
def get_sizes_range(self):
    self.sizes = [
        int(SB.side_bar.get_sprite(SB.START_ATTRACTORS_INPUT).value),
        int(SB.side_bar.get_sprite(SB.END_ATTRACTORS_INPUT).value)
    ]

def get_binary_chains(self):
    if len(self.binary_chains) < self.sizes[1]:
        self.binary_chains = Attractors.generate_binary_chains(self.sizes[1],self.binary_chains)

def conversion_matrix_bin_decimal(self,power,previous_matrix=np.array([1])):
    """Returns a matrix with value of each power value in the conversion from binary to decimal

    Parameters
    -----
    power : int
        Number of positions power to be calculated

    Returns
    -----

```

```

matrix : array
    Array that can be multiplied with the objective array to be converted into decimal values
"""
matrix = np.array([],np.uint16)
if power > previous_matrix.size:
    matrix = np.zeros((power-previous_matrix.size),np.uint16)
    actual_power_value = 1 << previous_matrix.size
    # Loops through the matrix
    for i in range(1,matrix.size+1):
        # print('actual power value >>',actual_power_value)
        matrix[-i] = int(actual_power_value)
        actual_power_value <<= 1
    # else: print('<-- The conversion matrix is bigger than the required power --->')
return np.concatenate((matrix,previous_matrix))

#-----
# Attractors
#-----
def compute_attractors(self):
    print('<-- Preparing to start the compute --->')
    self.get_sizes_range(); self.get_binary_chains()
    # Copy of the binary list that will be removed through the appearances in the compute process
    computed_chains = list(self.binary_chains)
    # Center of the evolution row
    center_cell = int(ECA.eca.number_cells_1D/2)
    index_hex_colors = 0
    print('<-- Starting attractors compute --->')
    starting_time = time.time()

    # Loop for every chains power universe
    for universe in range(self.sizes[0]-1, self.sizes[1]):
        print('<-- Computing attractors for the universe power {} --->'.format(universe+1))
        self.conversion_matrix = self.conversion_matrix_bin_decimal(universe+1, self.conversion_matrix)
        starting_cell = center_cell - int((universe+1)/2)
        # print('index chains >>',len(self.binary_chains[universe]))
        window_range = (starting_cell+1,starting_cell+universe+2)
        # list of the relations
        relations = []

        # Loop through the available chains of the universe
        for index_chains in range(len(self.binary_chains[universe])):

            # Validates it's still a string, meaning this node hasn't been explored yet in the three
            if type(computed_chains[universe][index_chains]) == str:

                # Loops through the characters of the binary chain
                updatables = []; previous_node = int(computed_chains[universe][index_chains],2)
                for index_value_chain in range(len(computed_chains[universe][index_chains])):
                    updatables = ECA.eca.update_cell_status(starting_cell+index_value_chain,True if
                        computed_chains[universe][index_chains][index_value_chain] == '1' else False,
                        invert=False, graphical_cell=True)
                    Grph.Graphics.graphics.updatable_elements.add(updatables[0]);
                    Grph.Graphics.graphics.updatable_rects.append(updatables[1])
                # As the chain has been used then it's no available anymore to be used
                computed_chains[universe][index_chains] = False

                # Keeps evolving until the appereance of an already used node or before appeared
                repeated_nodes = False
                while not repeated_nodes:
                    # Evolution of the actual chain in the window specified by the universe chains length
                    updatables =
                        ECA.eca.compute_next_generation(restrain_window=True,window_range=window_range)
                    Grph.Graphics.graphics.updatable_elements.add(updatables[1]);
                    Grph.Graphics.graphics.updatable_rects += updatables[0]
                # Chain resulted of the evolution

```

```

window_evolution = ECA.eca.get_actual_evolution_row(window_range)
chain_int = (window_evolution*self.conversion_matrix).sum()
""" print('Resulted chain integer >>',chain_int) """
relations.append((previous_node, chain_int))
# Verifies the chain hasn't appeared yet
if type(computed_chains[universe][chain_int]) == bool:
    repeated_nodes = True
    break
else:
    computed_chains[universe][chain_int] = False
    previous_node = chain_int
""" print('Computed chains >>', computed_chains) """
# When a three or a leaf has been found there's a jump in the evolution rows leaving a blank
# row to identify the different trees generated
ECA.eca.update_evolution_row(2)

# Generates the graph for the actual universe
self.generate_graph(universe+1,relations,index_hex_colors=index_hex_colors)
index_hex_colors += 1
if index_hex_colors >= len(HEX_COLORS_LIST): index_hex_colors = 0
print('\n<-- The compute of attractors finished in {} sec --->'.format(time.time()-starting_time))

def generate_graph(self,universe,relations,index_hex_colors:int=0):
    print('<-- Generating graph --->')
    # Creates the graph
    graph_nx = nx.DiGraph()
    graph_nx.add_edges_from(relations)
    graph = ig.Graph(directed=True); colours = []; labels = []
    # Adds the number of nodes
    graph.add_vertices(1<<universe)
    # Adds the nodes relations
    graph.add_edges(relations)
    # Adds the labels of the nodes
    labels.extend(list(np.arange(1<<universe)))
    # Adds the colors of the trees in this universe
    colours.extend([ HEX_COLORS_LIST[index_hex_colors] for _ in range(1<<universe) ])
    # Recovers the cycles
    print('Number nodes >>',1<<universe)
    for cycle in nx.simple_cycles(graph_nx):
        # 1st element: Number predecessors
        # 2nd element: Number of node
        more_predecessors_node = [-1,0]
        for node in cycle:
            sum_predecessors = sum(neighbour for neighbour in graph_nx.predecessors(node))
            if more_predecessors_node[0] < sum_predecessors: more_predecessors_node = [sum_predecessors,node]
        colours[more_predecessors_node[1]] = COLOUR_ATTRACTOR
        print('Attractor >> ', more_predecessors_node[1])

    # Adds the labels and colors of each vertex
    graph.vs['label'] = labels
    graph.vs['color'] = colours
    # Plots the graph
    graph_name = "./graphs/graph1_universe_{}.eps".format(universe)
    visual_style = {}
    # Set bbox and margin
    image_size = self.sizes_igraph[universe]
    visual_style["bbox"] = (image_size,image_size)
    visual_style["margin"] = 25
    # Set vertex properties
    visual_style["vertex_size"] = 20
    visual_style["vertex_label_size"] = 8
    # Don't curve the edges
    visual_style["edge_curved"] = False
    # Layout
    """ if universe > 9:

```

```

    visual_style["layout"] = graph.layout('rt_circular') """
# Plot the graph
ig.plot(graph, graph_name, **visual_style)
print('<-->'.format((image_size,image_size)))
print('<-->'.format(graph_name))

# Code intended to remove the topological alike trees of the attractors process
# Nevertheless the second library considered(networkx) wasn't that useful to make the image of the graph
# even though it's useful for analysing the graph, thus this code wasn't implemented due to lack of
# knowledge of the behaviour of other rules beside 110
""" def compute_attractors_2(self):
    print('<--> Preparing to start the compute ---')
    self.get_sizes_range(); self.get_binary_chains()
    # Copy of the binary list that will be removed through the appearances in the compute process
    computed_chains = list(self.binary_chains)
    # Center of the evolution row
    center_cell = int(ECA.eca.number_cells_1D/2)
    index_hex_colors = 0
    print('<--> Starting attractors compute ---')

    # Loop for every chains power universe
    for universe in range(self.sizes[0]-1, self.sizes[1]):
        print('<--> Computing attractors for the universe power {} ---'.format(universe+1))
        self.conversion_matrix = self.conversion_matrix_bin_decimal(universe+1, self.conversion_matrix)
        starting_cell = center_cell - int((universe+1)/2)
        # print('index chains >>',len(self.binary_chains[universe]))
        window_range = (starting_cell+1,starting_cell+universe+2)
        # list of the relations
        relations = []
        # Mask used to shorten the values of bit string into lenght of the actual universe
        bit_mask = 0; len_universe = len(computed_chains[universe])
        for _ in range(universe):
            bit_mask += 1; bit_mask <<= 1
        print('<--> Bit mask: {}'.format(bit_mask))

        # Loop through the available chains of the universe
        for index_chains in range(len(self.binary_chains[universe])):
            # Validates it's still a string, meaning this node hasn't been explored yet in the three
            if type(computed_chains[universe][index_chains]) == str:

                # print('index chain >> ', index_chains)
                # Loops through the characters of the binary chain
                updatables = []; previous_node = int(computed_chains[universe][index_chains],2)
                for index_value_chain in range(len(computed_chains[universe][index_chains])):
                    updatables = ECA.eca.update_cell_status(starting_cell+index_value_chain,True if
                        computed_chains[universe][index_chains][index_value_chain] == '1' else False,
                        invert=False, graphical_cell=True)
                    Grph.Graphics.graphics.updatable_elements.add(updatables[0]);
                    Grph.Graphics.graphics.updatable_rects.append(updatables[1])

                # As the chain has been used then it's no available anymore to be used
                computed_chains[universe][index_chains] = False
                # Verifying the topological alike versions of the actual starting node
                for index in self.shift_derived_nodes(index_chains,bit_mask,len_universe):
                    # Noticed that a derived node by shifting need no computing
                    if type(computed_chains[universe][index]) == str:
                        # print('Derived chain from initial node also removed from computation: ',index)
                        computed_chains[universe][index] = False

                # Keeps evolving until the appereance of an already used node or before appeared
                repeated_nodes = False
                while not repeated_nodes:
                    # Evolution of the actual chain in the window specified by the universe chains length
                    updatables =

```

```

        ECA.eca.compute_next_generation(restrain_window=True,window_range=window_range)
        Grph.Graphics.graphics.updatable_elements.add(updatables[1]);
        Grph.Graphics.graphics.updatable_rects += updatables[0]
    # Chain resulted of the evolution
    window_evolution = ECA.eca.get_actual_evolution_row(window_range)
    chain_int = (window_evolution*self.conversion_matrix).sum()
    # print('Resulted chain integer >>',chain_int)

    # Verifiying the topological alike versions of the resultant chain
    for index in self.shift_derived_nodes(chain_int,bit_mask,len_universe):
        # Noticed that a derived node by shifting need no computing
        if type(computed_chains[universe][index]) == str:
            # print('Derived chain also removed from computation: ',index)
            computed_chains[universe][index] = False

    relations.append([previous_node, chain_int])

    # Verifies the chain hasn't appeared yet
    if type(computed_chains[universe][chain_int]) == bool:
        repeated_nodes = True
        break
    else:
        computed_chains[universe][chain_int] = False
        previous_node = chain_int
    # When a three or a leaf has been found there's a jump in the evolution rows leaving a blank
    # row to identify the different trees generated
    ECA.eca.update_evolution_row(2)
# Generates the graph for the actual universe
# self.generate_graph(universe+1,index_hex_colors=index_hex_colors,ending_universe=8)
self.actual_universe = universe+1
print('relations >> ', relations)
shells = self.define_shell_leves(relations)
self.graph_attractors(relations,universe+1,shells)
# self.generate_graph(universe+1,index_hex_colors=index_hex_colors,ending_universe=8)
index_hex_colors += 1
if index_hex_colors >= len(HEX_COLORS_LIST): index_hex_colors = 0

def shift_derived_nodes(self,node,bit_mask,len_universe):
    aux_node = node
    derived_nodes = []
    # Moving left
    while aux_node:
        aux_node = (aux_node >> 1) & bit_mask
        if aux_node < len_universe: derived_nodes.append(int(aux_node))
    return derived_nodes

def define_shell_leves(self,relations):
    uniques_nodes,nodes_count = np.unique(relations,return_counts=True)
    shells, previous_level_nodes, level_nodes = [],set(),set(); aux_relations = list(relations)
    # First level only roots of threes
    for index_count in range(nodes_count.size):
        # As the node has only been referenced once means it's the root of a three
        if nodes_count[index_count] == 1: level_nodes.add(uniques_nodes[index_count])
    print('Roots >> ', len(level_nodes), level_nodes)

    # Loops until no more nodes get added for the shells levels
    more_levels = True
    while more_levels:
        shells.append(list(level_nodes)); previous_level_nodes = list(level_nodes)
        level_nodes.clear(); index_relations = 0

        while index_relations < len(aux_relations):
            if aux_relations[index_relations][0] in previous_level_nodes:
                level_nodes.add(aux_relations[index_relations][1])
            del aux_relations[index_relations]

```

```

        else: index_relations += 1

    more_levels = bool(level_nodes)

    # The left relations correspond to loops
    if aux_relations:
        for relation in aux_relations:
            shells[0].append(relation[0])
            self.list_attractors.append(relation[0])
    return shells

def graph_attractors(self,relations,universe,shells=None):
    mpl.use('Agg')
    plt.figure(figsize=self.sizes_graph[universe])

    graph = nx.DiGraph()
    graph.add_edges_from(relations,color='red')

    for cycle in nx.simple_cycles(graph):
        print('Cycles >> ', cycle)

    #pos = nx.kamada_kawai_layout(graph)
    #pos = nx.shell_layout(graph, shells,rotate=90,scale=2.5)
    pos = nx.spring_layout(graph,1/sqrt(len(relations)/2))
    #pos = nx.circular_layout(graph)
    #pos = nx.planar_layout(graph)
    #pos = nx.spiral_layout(graph)
    nx.draw_networkx_nodes(graph, pos)
    nx.draw_networkx_edges(graph, pos)
    nx.draw_networkx_labels(graph, pos)
    plt.savefig("./graphs/graph2_universe_{}.png".format(universe)) """

```

5.2 main.py

Archivo Python principal de la simulación, en este se maneja el ciclo principal de la aplicación gráfica, se crean e inician los flujos alternos en forma de hilos, la creación de los espacios gráficos y lógicos, constantes de configuración general del programa.

```

import os
import sys
# Preventing the welcome message of Python to be shown
os.environ['PYGAME_HIDE_SUPPORT_PROMPT'] = "hide"
from pygame import *
import threading
import queue
# User modules
import Graphics as gr
from Constants import *
from Layouts import Grid,BottomBar,SideBar
from GraphicalComponents import Text

#=====
# Configuration parameters
#=====
# WINDOW_MIN_WIDHT = 580
DELAY_GENERATIONS = 0.005 # Delay in seconds
# Colors of cells
ALIVE_COLOR = BLUE
DEAD_COLOR = DARK_BLACK
# Number of elements by side in the grid
# the total number of cell is NUMBER_CELLS_1D x NUMBER_EVOLUTIONS_GRID
GRID_WIDTH = 1250
GRID_PADDING = 0

```

```

NUMBER_CELLS_1D = 1000
NUMBER_EVOLUTIONS_GRID = 900
USE_MIN_SIZE = False
# Elements of the interface
SIDE_BAR_WIDTH = 250
BOTTOM_BAR_HEIGHT = 25
WINDOW_TITLE = 'Elementary Cellular Automaton'

=====
# Dynamic execution variables
=====

window = None
graphics = None
# Graphical elements
bottom_bar = None
side_bar = None

=====
# Configuration functions
=====

def waiting_frame(window_size):
    global window
    # window background
    window.fill(LIGHT_BLACK_2)
    window_size_center = window_size/2
    # Messages texts
    message = 'Creating the evolution space ...'
    text_message = Text(
        position=(window_size_center[0]-(BIG_FONT.size(message)[0])/2,window_size_center[1]-250),
        text=message,
        text_color=YELLOW,
        font=BIG_FONT)
    message_2 = '(Please wait a little)'
    text_message_2 = Text(
        position=(window_size_center[0]-(MEDIUM_FONT.size(message_2)[0])/2,window_size_center[1]-200),
        text=message_2,
        text_color=WHITE,
        font=MEDIUM_FONT)
    # Cell icon
    image_size = window_size_center[1]*.75
    icon = pygame.transform.scale(pygame.image.load('./images/cells_w.png'),(image_size,image_size))
    # Draws the messages and the icon cell in the screen
    text_message.print(window)
    text_message_2.print(window)
    window.blit(icon,(window_size_center[0]-image_size/2,window_size_center[1]-image_size/3))
    # Updates the whole display
    pygame.display.flip()

def first_draw_elements():
    global window,graphics
    global bottom_bar,side_bar
    # window background
    window.fill(LIGHT_BLACK_2)
    graphics.pointer.update()
    graphics.graphical_cells.draw(window)
    # Graphical elements
    bottom_bar.draw(window)
    side_bar_elements = pygame.sprite.Group(side_bar.get_graphical_sprites()); side_bar_elements.update()
    side_bar.draw(window)
    # Updates the whole display
    pygame.display.flip()

=====
# Main ECA programm
=====
```

```

def main():
    global window
    global graphics
    global bottom_bar,side_bar
    global GRID_WIDTH,GRID_PADDING,NUMBER_CELLS_1D,NUMBER_EVOLUTIONS_GRID,USE_MIN_SIZE
    #-----
    # Command Line Arguments
    #-----
    if len(sys.argv) == 3:
        NUMBER_CELLS_1D,NUMBER_EVOLUTIONS_GRID = [int(sys.argv[i]) for i in range(1,len(sys.argv))]
    if len(sys.argv) == 4:
        GRID_WIDTH,NUMBER_CELLS_1D,NUMBER_EVOLUTIONS_GRID = [int(sys.argv[i]) for i in range(1,len(sys.argv))]
    if len(sys.argv) == 5:
        GRID_WIDTH,NUMBER_CELLS_1D,NUMBER_EVOLUTIONS_GRID,GRID_PADDING = [int(sys.argv[i]) for i in
            range(1,len(sys.argv))]
    if len(sys.argv) == 6:
        GRID_WIDTH,NUMBER_CELLS_1D,NUMBER_EVOLUTIONS_GRID,GRID_PADDING,USE_MIN_SIZE = [int(sys.argv[i]) for i
            in range(1,len(sys.argv))]
    #-----
    # Pygame configurations
    #-----
    pygame.init()
    clock = pygame.time.Clock()
    #-----
    # Getting sizes from grid
    #-----
    grid_size,grid =
        Grid.optimal_grid_size((GRID_WIDTH-SIDE_BAR_WIDTH,GRID_WIDTH-BOTTOM_BAR_HEIGHT),NUMBER_CELLS_1D,NUMBER_EVOLUTIONS_GRID)
    #-----
    # Window configuration
    #-----
    window_display = 0
    window_size = grid_size + array([SIDE_BAR_WIDTH,BOTTOM_BAR_HEIGHT])
    print('--- Window size {} ---'.format(window_size))
    window = pygame.display.set_mode(window_size,display=window_display)
    pygame.display.set_caption(WINDOW_TITLE)
    # Setting the cells colors
    gr.Graphics.set_cells_colors([DEAD_COLOR,ALIVE_COLOR])
    #-----
    # Graphical elements
    #-----
    bottom_bar =
        BottomBar(LIGHT_BLACK_1,BOTTOM_BAR_HEIGHT,number_cells_1D=NUMBER_CELLS_1D,number_rows=NUMBER_EVOLUTIONS_GRID)
    bottom_bar.update_generations(0)
    side_bar = SideBar(DARK_BLACK,SIDE_BAR_WIDTH)
    #-----
    # Other Configurations
    #-----
    # Finishing the grid configuration
    grid.set_window(window)
    waiting_frame(window_size)
    #-----
    # Graphics
    #-----
    graphics = gr.Graphics(grid,side_bar=side_bar,bottom_bar=bottom_bar)
    graphics.set_delay_generations(DELAY_GENERATIONS)
    #-----
    # Cells
    #-----
    grid.locate_elements(graphics.graphical_cells.sprites())
    #-----
    # Pygame thread functions
    #-----
    queue_main = queue.Queue(); queue_refresh = queue.Queue()
    events_thread = threading.Thread(target=graphics.look_for_events)

```

```
actions_thread = threading.Thread(target=graphics.events_actions)
functions_thread = threading.Thread(target=graphics.execution_of_functions,args=[queue_main])
# Threads starting
events_thread.start()
actions_thread.start()
functions_thread.start()

#-----
# Main PyGame loop
#-----
first_draw_elements()
while not graphics.done:
    # Frames per Second
    clock.tick(45)

    graphics.draw_elements(window)

    # The queues are used to share objects to the main thread
    # In this case is used to execute plotting functions
    # of matplotlib from the main thread
    if not queue_main.empty():
        queue_main.get()()

pygame.quit()

if __name__ == '__main__': main()
```
