Instituto Politécnico Nacional

Escuela Superior de Cómputo

Teoría de la Computación

Programa Tablero

Profesor: Dr. Juarez Martinez Genaro

Alumno: Chen Yangfeng

chen1436915478@gmail.com

GRUPO
5BM2

13 de octubre de 2024

Índice

1. Introducción

En esta práctica, se desarrollará un programa de simulación de un juego basado en movimientos en un tablero de ajedrez de dimensiones reducidas (5x5). El objetivo es implementar un modelo que permita simular los movimientos de dos jugadores en el tablero, siguiendo reglas específicas de desplazamiento y configuraciones de juego. El programa deberá ser capaz de ejecutarse en modo automático o manual, permitiendo la interacción del usuario para introducir movimientos o generarlos de manera aleatoria. Además, se deberá garantizar la correcta visualización del progreso en el tablero, así como la representación gráfica de la red de movimientos generada.

El juego plantea un desafío adicional al requerir que los movimientos se realicen bajo un conjunto de reglas predefinidas, permitiendo tanto movimientos ortogonales como diagonales. Cada jugador tiene un punto de inicio y un objetivo final dentro del tablero. El primer jugador comenzará en la posición 1 (esquina superior izquierda), con la meta de alcanzar la posición 25 (esquina inferior derecha). El segundo jugador, en cambio, iniciará en la posición 5 (esquina superior derecha) y buscará llegar a la posición 21 (esquina inferior izquierda).

Para añadir dinamismo al juego, se establecerá aleatoriamente qué jugador empieza, y se generarán archivos con todos los movimientos posibles, así como con las rutas ganadoras de cada pieza, que servirán para la reconfiguración de las rutas en caso de que no se puedan avanzar más. La práctica también incluye la representación gráfica de los movimientos y de la red de estados generada por ambos jugadores, y se establecerá un número máximo de movimientos (entre 5 y 100) para finalizar la partida.

El objetivo final es crear un programa que no solo sea capaz de simular el juego y graficar los resultados, sino también de explorar la teoría de autómatas finitos no deterministas (NFA), representando los movimientos y posibles rutas en el tablero como una red de estados. Esta simulación permitirá a los estudiantes aplicar conceptos de teoría de autómatas, programación, y algoritmos de búsqueda de rutas, ofreciendo una forma visual y práctica de comprender estos temas complejos.

En el reporte final, se incluirá tanto el código fuente del programa como la explicación detallada del desarrollo del mismo, los archivos generados con las rutas de los jugadores, y las gráficas correspondientes.

2. Marco Teórico

2.1. Teoría de Autómatas Finitos

La teoría de autómatas es una rama de la informática que estudia los autómatas, los cuales son modelos matemáticos de máquinas abstractas que pueden estar en un estado determinado, cambiar a otros estados en función de las entradas recibidas y seguir reglas de transición. En este proyecto, se trabaja con un autómata finito no determinista (NFA), en el cual, para un estado dado y una entrada específica, pueden existir múltiples transiciones hacia diferentes estados.

Un NFA se puede representar mediante un grafo dirigido, donde los nodos corresponden a los estados y las aristas representan las transiciones posibles. En este contexto, el tablero de ajedrez de 5x5 será modelado como un NFA, donde cada casilla del tablero será un estado, y los movimientos válidos (ortogonales y diagonales) se representarán como transiciones entre estados.

2.2. Movimientos Ortogonales y Diagonales en el Tablero

En un tablero de 5x5, los movimientos ortogonales son aquellos que permiten desplazarse en las cuatro direcciones cardinales: arriba, abajo, izquierda y derecha. Por otro lado, los movimientos diagonales permiten moverse en las cuatro direcciones diagonales: arriba a la derecha, arriba a la izquierda, abajo a la derecha y abajo a la izquierda. Cada casilla se puede ver como un nodo en una red, y los movimientos posibles entre casillas definen las conexiones (transiciones) entre estos nodos.

Dado que se busca representar los movimientos como transiciones en un NFA, se considera que desde cada posición es posible moverse a otra siguiendo las reglas de los movimientos ortogonales y diagonales. El número de movimientos posibles estará limitado por la posición actual en el tablero, ya que en los bordes y esquinas algunas direcciones no estarán disponibles.

2.3. Representación Gráfica y Visualización

La visualización de los movimientos y de la red generada por las transiciones de ambos jugadores es esencial para comprender el comportamiento del sistema. En este proyecto, se deben graficar tanto el tablero con los movimientos realizados por los jugadores como la red completa de transiciones (NFA) que representa todos los estados y movimientos posibles.

La representación gráfica del tablero se puede realizar mediante bibliotecas de programación que soporten gráficos, como matplotlib en Python. Para graficar la red de estados, se puede utilizar una biblioteca de grafos como networkx, que permite la visualización de grafos dirigidos con nodos y aristas.

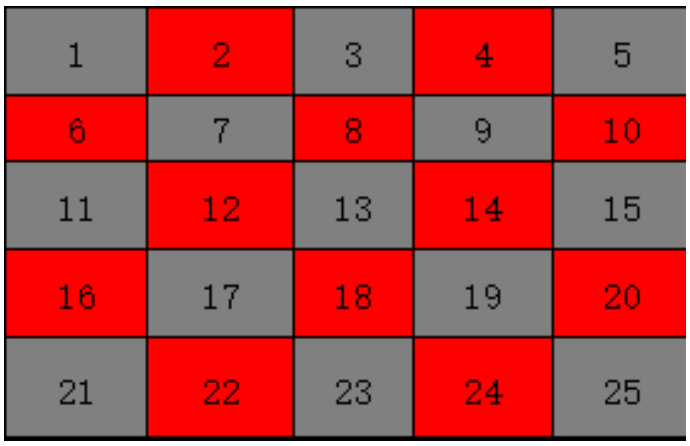

Figura 1: Imagen del tablero 5x5.

3. Desarrollo

3.1. Parte lógica del NFA del *Tablero*

3.1.1. Librerías Utilizadas

- ***NumPy*** (`import numpy as np`)
  Utilizada para manejar arreglos y configuraciones de gráficos. Se usa específicamente para establecer los límites y ticks en el tablero.

- ***Matplotlib*** (`import matplotlib.pyplot as plt`)
  Utilizada para crear gráficos y animaciones. Permite la visualización de los movimientos de los jugadores en el tablero y la representación gráfica de los nodos y sus conexiones.

- ***Random*** (`import random`)
  Utilizada para la generación de cadenas aleatorias de movimientos ('r' y 'b'), que simulan las posibles transiciones de los jugadores en el tablero.

- ***FuncAnimation*** (`from matplotlib.animation import FuncAnimation`)
  Empleada para crear animaciones de los movimientos de los jugadores. Permite actualizar el gráfico en cada iteración, mostrando el progreso de los jugadores.

- ***NetworkX*** (`import networkx as nx`)
  Utilizada para crear y visualizar grafos. Permite la representación de las rutas de los jugadores como un grafo dirigido, con nodos y aristas que indican las transiciones de estado.

```
1    import matplotlib.pyplot as plt
2    import numpy as np
3    import random
4    from matplotlib.animation import FuncAnimation
5    import networkx as nx
```

3.1.2. Tabla de Transición del tablero 5x5 con números pares color rojo y números impares color negro

Aquí tenemos la tabla de transición para nuestro programa NFA del tablero 5x5.

```
boardstate = {
1:  {'r': [2, 6], 'b': [7]},
2:  {'r': [6, 8], 'b': [1, 3, 7]},
3:  {'r': [2, 8, 4], 'b': [7, 9]},
4:  {'r': [8, 10], 'b': [3, 5, 9]},
5:  {'r': [4, 10], 'b': [9]},

6:  {'r': [2, 12], 'b': [1, 7, 11]},
7:  {'r': [2, 6, 8, 12], 'b': [1, 3, 11, 13]},
8:  {'r': [2, 4, 12, 14], 'b': [3, 7, 9, 13]},
9:  {'r': [4, 8, 10, 14], 'b': [3, 5, 13, 15]},
10: {'r': [4, 14], 'b': [5, 9, 15]},

11: {'r': [6, 12, 16], 'b': [7, 17]},
12: {'r': [6, 8, 16, 18], 'b': [7, 11, 13, 17]},
13: {'r': [8, 12, 14, 18], 'b': [7, 9, 17, 19]},
14: {'r': [8, 10, 18, 20], 'b': [9, 13, 15, 19]},
15: {'r': [10, 14, 20], 'b': [9, 19]},

16: {'r': [12, 22], 'b': [11, 17, 21]},
17: {'r': [12, 16, 18, 22], 'b': [11, 13, 21, 23]},
18: {'r': [12, 14, 22, 24], 'b': [13, 17, 19, 23]},
19: {'r': [14, 18, 20, 24], 'b': [13, 15, 23, 25]},
20: {'r': [14, 24], 'b': [15, 19, 25]},

21: {'r': [16, 22], 'b': [17]},
22: {'r': [16, 18], 'b': [17, 21, 23]},
23: {'r': [18, 22, 24], 'b': [17, 19]},
24: {'r': [18, 20], 'b': [19, 23, 25]},
25: {'r': [20, 24], 'b': [19]},
}
```

3.1.3. Transición en NFA

La función *nfa_transition(state, char)* devuelve los estados posibles a partir del estado actual (*state*), según el carácter de entrada (*char*), que puede ser 'r' o 'b'. Utiliza la variable *boardstate* para obtener las transiciones del autómata finito no determinista (NFA). La función toma un estado actual y un carácter de entrada, y devuelve una lista de estados alcanzables desde el estado actual para dicho carácter.

```
def nfa_transition(state, char):
return boardstate.get(state, {}).get(char, [])
```

3.1.4. Encontrar todas las rutas

La función *find_all_paths(current_states, input_string, current_path, all_paths)* encuentra recursivamente todas las rutas posibles a través del NFA, dados los estados actuales y una cadena de entrada. Si la cadena de entrada está vacía, la ruta actual se agrega a la lista de todas las rutas. Si no, se evalúan las transiciones para cada estado actual y se llaman recursivamente para los siguientes estados.

Parámetros:

- `current_states`: Lista de estados actuales desde donde se inician las transiciones.
- `input_string`: La cadena de entrada que queda por procesar.
- `current_path`: La ruta actual recorrida en el NFA.
- `all_paths`: Lista que acumula todas las rutas posibles.

```
    def find_all_paths(current_states, input_string, current_path, all_paths):
        if not input_string:
            all_paths.append(current_path)
            return

        char = input_string[0]
        next_input = input_string[1:]

        for state in current_states:
            next_states = nfa_transition(state, char)
            for next_state in next_states:
                find_all_paths([next_state], next_input, current_path + [next_state],
                    all_paths)
```

3.1.5. Buscar todas las rutas

La función *nfa_find_all_paths(input_string, start_state=1)* encuentra todas las rutas que el NFA puede seguir para una cadena de entrada dada, comenzando desde un estado inicial. Llama a *find_all_paths* para explorar recursivamente todas las rutas posibles y devuelve una lista con todas las rutas encontradas.

```
    def nfa_find_all_paths(input_string, start_state=1):
        current_states = [start_state]
        all_paths = []
        find_all_paths(current_states, input_string, [start_state], all_paths)
        return all_paths
```

3.1.6. Escribir rutas en archivo

La función *write_paths_to_file(paths, filename)* escribe una lista de rutas en un archivo de texto. Cada ruta es una secuencia de estados que se representa como una línea en el archivo, con los estados separados por comas.

```
    def write_paths_to_file(paths, filename):
        with open(filename, 'w') as file:
            for path in paths:
                file.write(','.join(map(str, path)) + '\n')
```

3.1.7. Verificar rutas

La función *verify_routes(filename, write_win_path=, target_state=9)* verifica si las rutas almacenadas en un archivo contienen un estado final específico (`target_state`). Si una ruta contiene el estado final, se elimina cualquier aparición posterior de este estado y se guarda la ruta modificada en otro archivo.

```
    def verify_routes(filename="Bloque_1\\Tablero\\routes.txt",
        write_win_path="Bloque_1\\Tablero\\win_routes.txt", target_state=9):
        valid_routes = []

        try:
            with open(filename, 'r') as file:
                routes = file.readlines()

                for route in routes:
                    path = list(map(int, route.strip().split(',')))
                    if target_state in path:
                        new_path = []
                        found_target = False

                        for state in path:
                            if state == target_state:
                                if not found_target:
                                    new_path.append(state)
                                    found_target = True
                            else:
```

```
                                new_path.append(state)

                        valid_routes.append(new_path)

            if valid_routes:
                write_paths_to_file(valid_routes, write_win_path)
            else:
                print(f"No hay rutas que contengan el estado final {target_state}.")

        except FileNotFoundError:
            print(f"Archivo '{filename}' no encontrado.")
```

3.1.8. Generar cadena aleatoria

La función *generar_cadena_rb()* genera aleatoriamente una cadena de caracteres `'r'` y `'b'` con una longitud entre 5 y 10 caracteres. Esta cadena se utiliza como entrada para explorar el NFA. Podemos modificar la variable *long_max* a 100, que nos pide en la practica. En este caso lo dejamos así para una buena prueba del código.

```
def generar_cadena_rb():
    long_min = 5
    long_max = 10
    longitud = random.randint(long_min, long_max)
    return ''.join(random.choice(['r', 'b']) for _ in range(longitud))
```

3.2. Gráfica del programa con los movimientos

3.2.1. Posiciones del tablero y Colores

El tablero se define en un diccionario `positions`, donde cada número del 1 al 25 tiene una posición (x, y) asociada. Un diccionario `colors` asigna un color a cada casilla (negro para números impares y rojo para números pares).

```
    positions = {
    1: (0, 4),  2: (1, 4),  3: (2, 4),  4: (3, 4),  5: (4, 4),
    6: (0, 3),  7: (1, 3),  8: (2, 3),  9: (3, 3), 10: (4, 3),
    11: (0, 2), 12: (1, 2), 13: (2, 2), 14: (3, 2), 15: (4, 2),
    16: (0, 1), 17: (1, 1), 18: (2, 1), 19: (3, 1), 20: (4, 1),
    21: (0, 0), 22: (1, 0), 23: (2, 0), 24: (3, 0), 25: (4, 0)
}

# Impares (negro), Pares (rojo)
colors = {i: 'black' if i % 2 != 0 else 'red' for i in range(1, 26)}
```

3.2.2. Lectura de Rutas desde Archivos

Esta función se encarga de leer las rutas de movimiento de cada jugador desde archivos de texto especificados en las rutas ruta_j1 y ruta_j2. Cada línea de los archivos contiene números separados por comas, que se almacenan en listas.

```
def leer_arrays_desde_txt(ruta_archivo):
    arrays = []
    with open(ruta_archivo, 'r') as archivo:
        for linea in archivo:
            numeros_str = linea.strip().split(',')
            numeros = [int(num) for num in numeros_str if num]
            if numeros:
                arrays.append(numeros)
    return arrays

rutas_jugador1 = leer_arrays_desde_txt(ruta_j1)
rutas_jugador2 = leer_arrays_desde_txt(ruta_j2)
```

3.2.3. Selección Aleatoria de Rutas

Se cargan las rutas para ambos jugadores desde sus respectivos archivos y se selecciona una ruta aleatoria para cada uno.

```
    ruta_j1 = random.choice(rutas_jugador1)
    ruta_j2 = random.choice(rutas_jugador2)

    pos_j1 = ruta_j1[0]
    pos_j2 = ruta_j2[0]
```

3.2.4. Configuración del Gráfico

Se utiliza la biblioteca Matplotlib para crear un gráfico de 5x5 donde se dibujan las casillas del tablero usando rectángulos. Cada casilla está etiquetada con su número correspondiente, lo que proporciona una representación visual clara del tablero de juego.

```
    fig, ax = plt.subplots(figsize=(6, 6))

    for i in range(1, 26):
        x, y = positions[i]
        ax.add_patch(plt.Rectangle((x - 0.5, y - 0.5), 1, 1, color=colors[i]))
        ax.text(x, y, str(i), color='white', fontsize=16, ha='center', va='center')

    ax.set_xticks(np.arange(-0.5, 5.5, 1))
    ax.set_yticks(np.arange(-0.5, 5.5, 1))
    ax.grid(True)
    plt.xlim(-0.5, 4.5)
    plt.ylim(-0.5, 4.5)
    ax.set_aspect('equal')

    # Inicializar las posiciones de los jugadores
    jugador1, = ax.plot([], [], 'bo', markersize=20, label='Jugador␣1')
    jugador2, = ax.plot([], [], 'go', markersize=20, label='Jugador␣2')
    titulo = ax.text(2, 5, "", ha="center", fontsize=14)

    # Estado inicial
    indice_j1 = 0
    indice_j2 = 0
    turno_j1 = random.choice([True, False])
    estado_final1 = 25
    estado_final2 = 21
    empate1 = False
    empate2 = False
```

3.2.5. Movimiento y Actualización de Jugadores

Esta función es el núcleo de la animación, gestionando el movimiento de los jugadores en cada turno. Dependiendo de a quién le toque, se mueve al jugador correspondiente y se verifica si hay colisiones con el otro jugador. En caso de colisión, se busca una ruta alternativa para el jugador afectado. Si no se encuentra una ruta alternativa, se notifica que el jugador cede su turno.

```
    def encontrar_ruta_alternativa(rutas_disponibles, ruta_actual, indice_actual):
        for ruta in rutas_disponibles:
            if ruta[:indice_actual] == ruta_actual[:indice_actual] and len(ruta) >
                indice_actual:
                # Verifica si la ruta se desv a despu s del punto de conflicto
                if ruta[indice_actual] != ruta_actual[indice_actual]:
                    return ruta
        return None

    def actualizar(i):
        nonlocal pos_j1, pos_j2, turno_j1, indice_j1, indice_j2, ruta_j1, ruta_j2,
            empate1, empate2

```

```
        if turno_j1:  # Turno de Jugador 1
            if indice_j1 < len(ruta_j1):
                siguiente_pos_j1 = ruta_j1[indice_j1]
                if siguiente_pos_j1 == pos_j2:  # Colisi n
                    nueva_ruta1 = encontrar_ruta_alternativa(rutas_jugador1, ruta_j1,
                        indice_j1)
                    if nueva_ruta1 is None:
                        titulo.set_text(f"  Colisin  ! Jugador 1 cede el turno")
                        turno_j1 = False
                        empate1 = True
                    else:
                        print(f"Ruta del jugador 1: {ruta_j1} -> nueva ruta jugador 1:
                            {nueva_ruta1}")
                        ruta_j1 = nueva_ruta1
                        turno_j1 = True
                        titulo.set_text(f"  Colisin  ! Jugador 1 cambia de ruta")
                else:
                    pos_j1 = siguiente_pos_j1
                    x_j1, y_j1 = positions[pos_j1]
                    jugador1.set_data([x_j1], [y_j1])
                    indice_j1 += 1
                    titulo.set_text(f"Jugador 1 se mueve a {pos_j1}")
                    turno_j1 = False

        else:  # Turno de Jugador 2
            if indice_j2 < len(ruta_j2):
                siguiente_pos_j2 = ruta_j2[indice_j2]
                if siguiente_pos_j2 == pos_j1:  # Colisi n
                    nueva_ruta2 = encontrar_ruta_alternativa(rutas_jugador2, ruta_j2,
                        indice_j2)
                    if nueva_ruta2 is None:
                        titulo.set_text(f"  Colisin  ! Jugador 2 cede el turno")
                        turno_j1 = True
                        empate2 = True
                    else:
                        print(f"Ruta del jugador 2: {ruta_j2} -> nueva ruta jugador 2:
                            {nueva_ruta2}")
                        ruta_j2 = nueva_ruta2
                        turno_j1 = False
                        titulo.set_text(f"  Colisin  ! Jugador 2 cambia de ruta")
                else:
                    pos_j2 = siguiente_pos_j2
                    x_j2, y_j2 = positions[pos_j2]
                    jugador2.set_data([x_j2], [y_j2])
                    indice_j2 += 1
                    titulo.set_text(f"Jugador 2 se mueve a {pos_j2}")
                    turno_j1 = True


        if pos_j1 == estado_final1:
            print(" Jugador  1 gan !")
            ani.event_source.stop()
            print(f"Ruta ganadora: {ruta_j1}")
        elif pos_j2 == estado_final2:
            print(" Jugador  2 gan !")
            ani.event_source.stop()
            print(f"Ruta ganadora: {ruta_j2}")
        elif empate1 and empate2:
            print("Empate")
            ani.event_source.stop()
            print(f"Ruta J1: {ruta_j1}, Ruta J2: {ruta_j2}")
```

3.3. Gráfica de la Red de los movimientos

La función `red` tiene como objetivo construir y visualizar un grafo dirigido basado en rutas definidas en un archivo de texto. Al inicio, se inicializa un objeto `DiGraph` de la biblioteca NetworkX, que representará las conexiones entre los nodos. La función abre el archivo especificado, leyendo las rutas que están separadas por comas en cada línea, y almacena cada ruta en una lista. A partir de la primera ruta, se determina la longitud total, lo cual define cuántas columnas tendrá el grafo.

Con la información de las rutas, la función procede a crear nodos y aristas. Cada nodo se define como una tupla que combina la posición y el estado correspondiente en la ruta. Si el nodo actual no es el último en su ruta, se crea una arista que conecta este nodo con el siguiente. Luego, para distribuir los nodos en el gráfico, se configuran sus posiciones, organizándolos en columnas basadas en su posición en la ruta y apilándolos verticalmente con un espacio determinado entre ellos.

Para la visualización, se utiliza la función `nx.draw`, que dibuja el grafo con las posiciones establecidas y ajusta el tamaño y el color de los nodos. Además, se generan etiquetas para los nodos que indican su estado. Los nodos que corresponden al estado final se destacan visualmente con un color diferente (cyan) y un tamaño reducido para resaltar su importancia en el grafo. Se incluye también un mensaje informativo en la parte superior del gráfico, que se pasa como argumento a la función, junto con un título que describe el contenido del gráfico. Finalmente, la visualización se completa con `plt.show()`, lo que permite ver las conexiones entre los nodos de manera clara y efectiva.

```
    def red(archivo, mensaje, estado_fin):

    G = nx.DiGraph()

    with open(archivo, 'r') as f:
        rutas = [line.strip().split(',') for line in f]
        longitud_ruta = len(rutas[0])


        posiciones = {}
        for idx, ruta in enumerate(rutas):
            for pos, estado in enumerate(ruta):
                nodo = (pos + 1, int(estado))
                if nodo not in posiciones:
                    posiciones[nodo] = (pos, -idx)
                if pos < longitud_ruta - 1:
                    nodo_siguiente = (pos + 2, int(ruta[pos + 1]))
                    G.add_edge(nodo, nodo_siguiente)


    plt.figure(figsize=(9, 8))
    num_columnas = longitud_ruta
    distancia_entre_columnas = 2

    for columna in range(1, num_columnas + 1):
        nodos_en_columna = [nodo for nodo in posiciones.keys() if nodo[0] == columna]

        for idx, nodo in enumerate(nodos_en_columna):
            posiciones[nodo] = (columna, -idx * distancia_entre_columnas)



    nx.draw(G, pos=posiciones, with_labels=False, node_size=800, node_color="lightblue",
        font_size=10, font_weight='bold', arrows=True)

    etiquetas = {nodo: f"{nodo[1]}" for nodo in G.nodes()}
    nx.draw_networkx_labels(G, pos=posiciones, labels=etiquetas, font_color='black',
        font_size=8)

    nodos_doble_circulo = [nodo for nodo in G.nodes() if nodo[1] == estado_fin]


    nx.draw(G, nodelist=nodos_doble_circulo, pos=posiciones, with_labels=False,
        node_size=600, node_color="cyan", font_size=10, font_weight='bold', arrows=True)



    plt.text(0.5, 0.95, mensaje, ha='center', va='center', fontsize=12,
        transform=plt.gca().transAxes)

    plt.title('Graph of Node Connections Across Positions')
    plt.axis('off')
    plt.show()
```

3.4. Función principal del Tablero

La función `main` es el punto de entrada principal para la ejecución del programa, estableciendo las condiciones iniciales y orquestando el flujo de la aplicación. Al inicio, se definen los estados iniciales y objetivos para dos jugadores, así como las rutas de archivo para almacenar las rutas generadas y las rutas ganadoras. Estos archivos contienen información crucial sobre los movimientos posibles de cada jugador en el juego.

El usuario tiene la opción de elegir entre un modo automático o manual para generar la cadena de movimientos. En el modo automático, se genera una cadena aleatoria de movimientos utilizando la función `generar_cadena_rb()`, la cual es luego mostrada al usuario. En el modo manual, se solicita al usuario que introduzca una cadena de movimientos de forma manual, proporcionando flexibilidad en cómo se desea jugar.

Posteriormente, se llama a la función `nfa_find_all_paths` para ambos jugadores, utilizando la cadena de movimientos generada y sus respectivos estados iniciales. Esto devuelve todas las rutas posibles que los jugadores pueden seguir basándose en los movimientos proporcionados. Las rutas generadas para cada jugador se escriben en archivos correspondientes mediante la función `write_paths_to_file`.

La función `verify_routes` se encarga de verificar las rutas generadas, comparándolas con las rutas ganadoras predefinidas, para determinar si los jugadores han alcanzado sus respectivos objetivos. Al finalizar, se informa al usuario sobre los movimientos escritos para ambos jugadores.

Finalmente, se ejecutan las funciones `run_game_animation` y `red` para visualizar el juego y sus rutas, mostrando así el desarrollo del mismo. La función `run_game_animation` ejecuta una animación del juego basado en las rutas ganadoras, mientras que `red` visualiza las conexiones de nodos en un grafo para cada jugador, resaltando sus respectivas rutas. La función concluye al comprobar que se está ejecutando como programa principal.

```
    def main():
        start_state1 = 1
        start_state2 = 5
        target1 = 25
        target2 = 21

        all_routes_player1 = "Bloque_1\\Tablero\\all_routes_player1.txt"
        all_routes_player2 = "Bloque_1\\Tablero\\all_routes_player2.txt"
        win_routes_player1 = "Bloque_1\\Tablero\\win_routes_player1.txt"
        win_routes_player2 = "Bloque_1\\Tablero\\win_routes_player2.txt"



        print("1. Modo autom tico")
        print("2. Modo manual")
        opcion = "1"
        opcion = input("Elige la opci n: ")
        if opcion == "1":
            cadena = generar_cadena_rb()
            print(f"La cadena de movimiento generada es: {cadena}")
        elif opcion == "2":
            cadena = str(input("Escribe la cadena de movimiento (ex. rbbbrb): "))

        player1 = nfa_find_all_paths(cadena, start_state1)
        player2 = nfa_find_all_paths(cadena, start_state2)

        write_paths_to_file(player1, all_routes_player1)
        write_paths_to_file(player2, all_routes_player2)
        verify_routes(all_routes_player1, win_routes_player1, target1)
        verify_routes(all_routes_player2, win_routes_player2, target2)
        print(f"Escribiendo los movimientos '{cadena}' para ambos jugadores.")


```

```
        run_game_animation(win_routes_player1, win_routes_player2)
        red(all_routes_player1, "Red del jugador 1", target1)
        red(all_routes_player2, "Red del jugador 2", target2)


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

4. Ejecución del Programa

Al ejecutar el programa, el usuario puede elegir el modo automático o el modo manual. Si el usuario elige el modo automático ingresando el numero 1, generara la cadena de movimiento aleatorio.

```
1. Modo automático
2. Modo manual
Elige la opción:
```

Figura 2: Menú para elegir el modo.

```
1. Modo automático
2. Modo manual
Elige la opción: 1
```

Figura 3: Ingresando el '1' para el modo automático.

```
La cadena de movimiento generada es: bbbbbbrbrr
Escribiendo los movimientos 'bbbbbbrbrr' para ambos jugadores.
```

Figura 4: Genera la cadena de movimiento aleatoriamente.

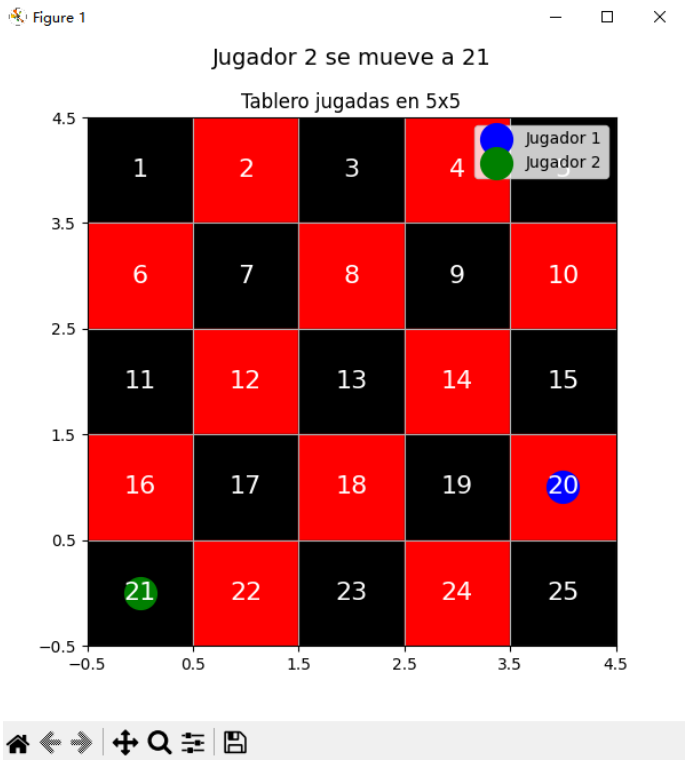

Figura 5: Muestra la animación de los dos jugadores.

```
Ruta del jugador 1: [1, 7, 13, 19, 25, 19, 15, 20, 15, 20, 14]
Ruta del jugador 2: [5, 9, 15, 19, 13, 17, 11, 16, 21, 22, 18]
Ruta del jugador 1: [1, 7, 13, 19, 25, 19, 15, 20, 15, 20, 14] -> nueva ruta jugador 1: [1, 7, 13, 7, 3, 9, 15, 20, 25, 20, 14]
¡Jugador 2 ganó!
Ruta ganadora: [5, 9, 15, 19, 13, 17, 11, 16, 21, 22, 18]
```

Figura 6: Cuando gana un jugador.

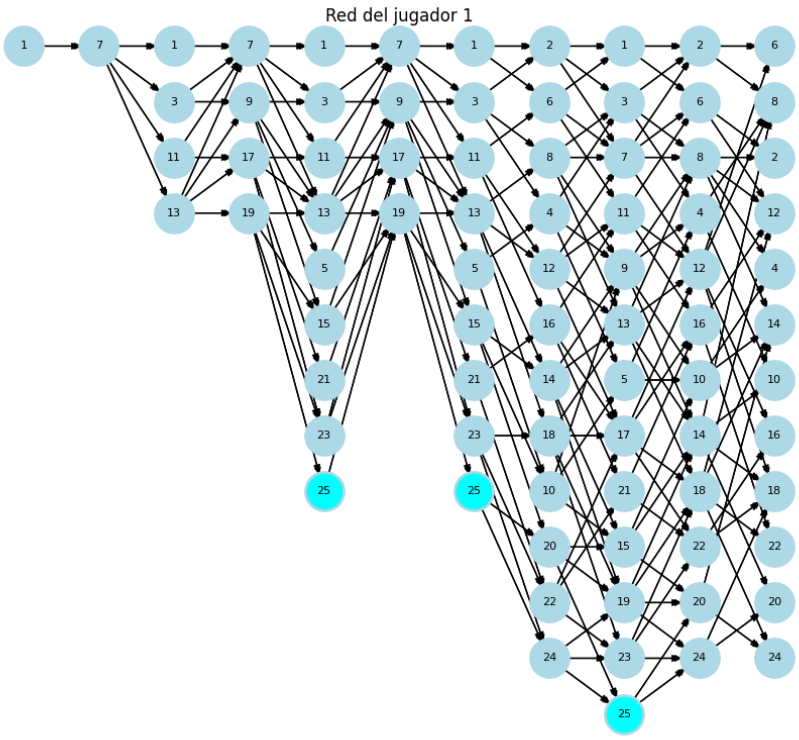

Figura 7: Red del NFA del jugador 1.

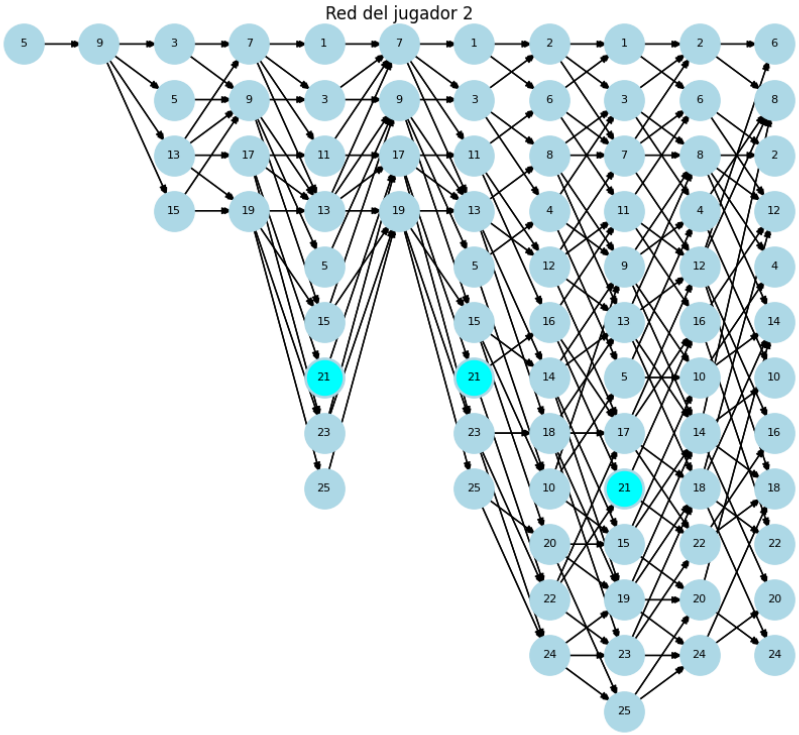

Figura 8: Red del NFA del jugador 2.

5. Conclusión

El código proporcionado implementa un juego basado en un autómata no determinista (NFA) en un tablero de 5x5, donde dos jugadores se mueven a través de diferentes posiciones en función de una cadena de caracteres que representan movimientos. Utiliza una estructura de datos que define cómo cada estado del tablero se relaciona con otros a través de movimientos rojos ('r') y azules ('b'). Los jugadores inician en posiciones específicas y deben seguir las rutas definidas por la cadena generada o ingresada. El programa permite tanto un modo automático, donde la cadena de movimiento se genera aleatoriamente, como un modo manual, donde el usuario puede introducir su propia cadena.

A través de la función `nfa_find_all_paths`, el código explora todas las rutas posibles que los jugadores pueden tomar según las reglas definidas en el `boardstate`. Posteriormente, las rutas se verifican para determinar si contienen los estados finales deseados. Si un jugador alcanza su estado final, el juego se detiene y se muestra la ruta ganadora. También se maneja la posibilidad de colisiones, donde si ambos jugadores intentan ocupar la misma posición, se les ofrece la oportunidad de cambiar a una nueva ruta que no cause la colisión.

Finalmente, el código incluye una representación visual del juego y de las rutas a través de animaciones y gráficos generados con Matplotlib y NetworkX. Esto no solo proporciona una representación clara de la dinámica del juego, sino que también permite visualizar las conexiones entre diferentes estados en forma de grafo. En conjunto, este código ofrece una implementación interesante de un juego de tablero interactivo que combina la teoría de autómatas y la visualización gráfica, permitiendo así un análisis visual de las estrategias de los jugadores y sus rutas en el tablero.

6. Referencias Bibliográficas

Referencias

[1] Hopcroft, J., Motwani, R., Ullman, J. (2001). *Introduction to Automata Theory, Languages, and Computation* https://www-2.dc.uba.ar/staff/becher/Hopcroft-Motwani-Ullman-2001.pdf

[2] Jeffrey D. Ullman. (2009). *CS154: Introduction to Automata and Complexity Theory* http://infolab.stanford.edu/~ullman/ialc/spr10/slides/fa3.ppt

7. Anexo

7.1. Código completo del *Tablero* implementado en Python

```
import matplotlib.pyplot as plt
import numpy as np
import random
from matplotlib.animation import FuncAnimation
import networkx as nx


# 3x3
# boardstate = {
#     1: {'r': [2, 4], 'b': [5]},
#     2: {'r': [4, 6], 'b': [1, 3, 5]},
#     3: {'r': [2, 6], 'b': [5]},
#     4: {'r': [2, 8], 'b': [1, 5, 7]},
#     5: {'r': [2, 4, 6, 8], 'b': [1, 3, 7, 9]},
#     6: {'r': [2, 8], 'b': [3, 5, 9]},
#     7: {'r': [4, 8], 'b': [5]},
#     8: {'r': [4, 6], 'b': [5, 7, 9]},
#     9: {'r': [6, 8], 'b': [5]}
# }


# 5x5
boardstate = {
    1:  {'r': [2, 6], 'b': [7]},
    2:  {'r': [6, 8], 'b': [1, 3, 7]},
    3:  {'r': [2, 8, 4], 'b': [7, 9]},
    4:  {'r': [8, 10], 'b': [3, 5, 9]},
    5:  {'r': [4, 10], 'b': [9]},

    6:  {'r': [2, 12], 'b': [1, 7, 11]},
    7:  {'r': [2, 6, 8, 12], 'b': [1, 3, 11, 13]},
    8:  {'r': [2, 4, 12, 14], 'b': [3, 7, 9, 13]},
    9:  {'r': [4, 8, 10, 14], 'b': [3, 5, 13, 15]},
    10: {'r': [4, 14], 'b': [5, 9, 15]},

    11: {'r': [6, 12, 16], 'b': [7, 17]},
    12: {'r': [6, 8, 16, 18], 'b': [7, 11, 13, 17]},
    13: {'r': [8, 12, 14, 18], 'b': [7, 9, 17, 19]},
    14: {'r': [8, 10, 18, 20], 'b': [9, 13, 15, 19]},
    15: {'r': [10, 14, 20], 'b': [9, 19]},

    16: {'r': [12, 22], 'b': [11, 17, 21]},
    17: {'r': [12, 16, 18, 22], 'b': [11, 13, 21, 23]},
    18: {'r': [12, 14, 22, 24], 'b': [13, 17, 19, 23]},
    19: {'r': [14, 18, 20, 24], 'b': [13, 15, 23, 25]},
    20: {'r': [14, 24], 'b': [15, 19, 25]},

    21: {'r': [16, 22], 'b': [17]},
    22: {'r': [16, 18], 'b': [17, 21, 23]},
    23: {'r': [18, 22, 24], 'b': [17, 19]},
    24: {'r': [18, 20], 'b': [19, 23, 25]},
```

```
        25: {'r': [20, 24], 'b': [19]},
    }



    def nfa_transition(state, char):
        """Devuelve los estados posibles a partir del estado actual seg n el car cter de entrada."""
        return boardstate.get(state, {}).get(char, [])


    def find_all_paths(current_states, input_string, current_path, all_paths):
        """recursivamente todas las rutas con la cadena de entrada."""
        if not input_string:  # If the input string is empty, add the current path to all paths
            all_paths.append(current_path)
            return

        char = input_string[0]
        next_input = input_string[1:]

        for state in current_states:
            next_states = nfa_transition(state, char)
            for next_state in next_states:
                # Recursively find paths from the next state
                find_all_paths([next_state], next_input, current_path + [next_state], all_paths)


    def nfa_find_all_paths(input_string, start_state=1):
        """Todas las rutas posibles"""
        current_states = [start_state]
        all_paths = []

        # Find all paths that the NFA can take for the input string
        find_all_paths(current_states, input_string, [start_state], all_paths)

        return all_paths


    def write_paths_to_file(paths, filename):
        with open(filename, 'w') as file:
            for path in paths:
                file.write(','.join(map(str, path)) + '\n')


    def verify_routes(filename="Bloque_1\\Tablero\\routes.txt", write_win_path="Bloque_1\\Tablero\\win_routes.txt", target_state=9):
        """Verificar si la ruta contiene el estado final"""
        valid_routes = []

        try:
            with open(filename, 'r') as file:
                routes = file.readlines()

                for route in routes:
                    path = list(map(int, route.strip().split(',')))
                    if target_state in path:
                        new_path = []
                        found_target = False

                        for state in path:
                            if state == target_state:
                                if not found_target:
                                    new_path.append(state)
                                    found_target = True
                            else:
                                new_path.append(state)

                        valid_routes.append(new_path)

            if valid_routes:
                write_paths_to_file(valid_routes, write_win_path)
            else:
                print(f"No hay rutas que contenga la ruta final {target_state}.")

```

```
        except FileNotFoundError:
            print(f"Archivo '{filename}' no encontrado.")


    def generar_cadena_rb():
        long_min = 5
        long_max = 10
        longitud = random.randint(long_min, long_max)
        return ''.join(random.choice(['r', 'b']) for _ in range(longitud))



    ########################################################################## GRAFICA
        ##############################################################################################

    def run_game_animation(ruta_j1, ruta_j2):
        positions = {
            1: (0, 4),  2: (1, 4),  3: (2, 4),  4: (3, 4),  5: (4, 4),
            6: (0, 3),  7: (1, 3),  8: (2, 3),  9: (3, 3), 10: (4, 3),
            11: (0, 2), 12: (1, 2), 13: (2, 2), 14: (3, 2), 15: (4, 2),
            16: (0, 1), 17: (1, 1), 18: (2, 1), 19: (3, 1), 20: (4, 1),
            21: (0, 0), 22: (1, 0), 23: (2, 0), 24: (3, 0), 25: (4, 0)
        }

        # Impares (negro), Pares (rojo)
        colors = {i: 'black' if i % 2 != 0 else 'red' for i in range(1, 26)}

        def leer_arrays_desde_txt(ruta_archivo):
            arrays = []
            with open(ruta_archivo, 'r') as archivo:
                for linea in archivo:
                    numeros_str = linea.strip().split(',')
                    numeros = [int(num) for num in numeros_str if num]
                    if numeros:
                        arrays.append(numeros)
            return arrays

        rutas_jugador1 = leer_arrays_desde_txt(ruta_j1)
        rutas_jugador2 = leer_arrays_desde_txt(ruta_j2)

        ruta_j1 = random.choice(rutas_jugador1)
        ruta_j2 = random.choice(rutas_jugador2)

        pos_j1 = ruta_j1[0]
        pos_j2 = ruta_j2[0]

        # Crear el gr fico
        fig, ax = plt.subplots(figsize=(6, 6))

        # Dibujar las casillas del tablero
        for i in range(1, 26):
            x, y = positions[i]
            ax.add_patch(plt.Rectangle((x - 0.5, y - 0.5), 1, 1, color=colors[i]))
            ax.text(x, y, str(i), color='white', fontsize=16, ha='center', va='center')

        # Configurar los ejes
        ax.set_xticks(np.arange(-0.5, 5.5, 1))
        ax.set_yticks(np.arange(-0.5, 5.5, 1))
        ax.grid(True)
        plt.xlim(-0.5, 4.5)
        plt.ylim(-0.5, 4.5)
        ax.set_aspect('equal')

        jugador1, = ax.plot([], [], 'bo', markersize=20, label='Jugador 1')
        jugador2, = ax.plot([], [], 'go', markersize=20, label='Jugador 2')
        titulo = ax.text(2, 5, "", ha="center", fontsize=14)

        indice_j1 = 0
        indice_j2 = 0
        turno_j1 = random.choice([True, False])
        estado_final1 = 25
        estado_final2 = 21
        empate1 = False
        empate2 = False

        def encontrar_ruta_alternativa(rutas_disponibles, ruta_actual, indice_actual):
```

```
199             for ruta in rutas_disponibles:
200                 if ruta[:indice_actual] == ruta_actual[:indice_actual] and len(ruta) >
                        indice_actual:
201                     # Verifica si la ruta se desv a despu s del punto de conflicto
202                     if ruta[indice_actual] != ruta_actual[indice_actual]:
203                         return ruta
204             return None
205 
206         def actualizar(i):
207             nonlocal pos_j1, pos_j2, turno_j1, indice_j1, indice_j2, ruta_j1, ruta_j2,
                    empate1, empate2
208 
209             if turno_j1:  # Turno de Jugador 1
210                 if indice_j1 < len(ruta_j1):
211                     siguiente_pos_j1 = ruta_j1[indice_j1]
212                     if siguiente_pos_j1 == pos_j2:  # Colisi n
213                         nueva_ruta1 = encontrar_ruta_alternativa(rutas_jugador1,
                                ruta_j1, indice_j1)
214                         if nueva_ruta1 is None:
215                             titulo.set_text(f"  Colisin  !␣Jugador␣1␣cede␣el␣turno")
216                             turno_j1 = False
217                             empate1 = True
218                         else:
219                             print(f"Ruta␣del␣jugador␣1:␣{ruta_j1}␣->␣nueva␣ruta␣jugador␣
                                    1:␣{nueva_ruta1}")
220                             ruta_j1 = nueva_ruta1
221                             turno_j1 = True
222                             titulo.set_text(f"  Colisin  !␣Jugador␣1␣cambia␣de␣ruta")
223                     else:
224                         pos_j1 = siguiente_pos_j1
225                         x_j1, y_j1 = positions[pos_j1]
226                         jugador1.set_data([x_j1], [y_j1])
227                         indice_j1 += 1
228                         titulo.set_text(f"Jugador␣1␣se␣mueve␣a␣{pos_j1}")
229                         turno_j1 = False
230 
231             else:  # Turno de Jugador 2
232                 if indice_j2 < len(ruta_j2):
233                     siguiente_pos_j2 = ruta_j2[indice_j2]
234                     if siguiente_pos_j2 == pos_j1:  # Colisi n
235                         nueva_ruta2 = encontrar_ruta_alternativa(rutas_jugador2,
                                ruta_j2, indice_j2)
236                         if nueva_ruta2 is None:
237                             titulo.set_text(f"  Colisin  !␣Jugador␣2␣cede␣el␣turno")
238                             turno_j1 = True
239                             empate2 = True
240                         else:
241                             print(f"Ruta␣del␣jugador␣2:␣{ruta_j2}␣->␣nueva␣ruta␣jugador␣
                                    2:␣{nueva_ruta2}")
242                             ruta_j2 = nueva_ruta2
243                             turno_j1 = False
244                             titulo.set_text(f"  Colisin  !␣Jugador␣2␣cambia␣de␣ruta")
245                     else:
246                         pos_j2 = siguiente_pos_j2
247                         x_j2, y_j2 = positions[pos_j2]
248                         jugador2.set_data([x_j2], [y_j2])
249                         indice_j2 += 1
250                         titulo.set_text(f"Jugador␣2␣se␣mueve␣a␣{pos_j2}")
251                         turno_j1 = True
252 
253 
254             if pos_j1 == estado_final1:
255                 print("  Jugador ␣1␣gan  !")
256                 ani.event_source.stop()
257                 print(f"Ruta␣ganadora:␣{ruta_j1}")
258             elif pos_j2 == estado_final2:
259                 print("  Jugador ␣2␣gan  !")
260                 ani.event_source.stop()
261                 print(f"Ruta␣ganadora:␣{ruta_j2}")
262             elif empate1 and empate2:
263                 print("Empate")
264                 ani.event_source.stop()
265                 print(f"Ruta␣J1:␣{ruta_j1},␣Ruta␣J2:␣{ruta_j2}")
266 
267         def init():
268             jugador1.set_data([], [])
```

```
            jugador2.set_data([], [])
            titulo.set_text("")
            return jugador1, jugador2, titulo

        ani = FuncAnimation(fig, actualizar, frames=50, init_func=init, interval=1000,
            repeat=False)
        print(f"Ruta del jugador 1: {ruta_j1}")
        print(f"Ruta del jugador 2: {ruta_j2}")

        plt.legend()
        plt.title("Tablero jugadas en 5x5")
        plt.show()


    ############################################################################  RED
        ##############################################################################

    def red(archivo, mensaje, estado_fin):

        G = nx.DiGraph()

        with open(archivo, 'r') as f:
            rutas = [line.strip().split(',') for line in f]
            longitud_ruta = len(rutas[0])


            posiciones = {}
            for idx, ruta in enumerate(rutas):
                for pos, estado in enumerate(ruta):
                    nodo = (pos + 1, int(estado))
                    if nodo not in posiciones:
                        posiciones[nodo] = (pos, -idx)
                    if pos < longitud_ruta - 1:
                        nodo_siguiente = (pos + 2, int(ruta[pos + 1]))
                        G.add_edge(nodo, nodo_siguiente)


        plt.figure(figsize=(9, 8))
        num_columnas = longitud_ruta
        distancia_entre_columnas = 2

        for columna in range(1, num_columnas + 1):
            nodos_en_columna = [nodo for nodo in posiciones.keys() if nodo[0] == columna]

            for idx, nodo in enumerate(nodos_en_columna):
                posiciones[nodo] = (columna, -idx * distancia_entre_columnas)



        nx.draw(G, pos=posiciones, with_labels=False, node_size=800,
            node_color="lightblue", font_size=10, font_weight='bold', arrows=True)

        etiquetas = {nodo: f"{nodo[1]}" for nodo in G.nodes()}
        nx.draw_networkx_labels(G, pos=posiciones, labels=etiquetas, font_color='black',
            font_size=8)

        nodos_doble_circulo = [nodo for nodo in G.nodes() if nodo[1] == estado_fin]


        nx.draw(G, nodelist=nodos_doble_circulo, pos=posiciones, with_labels=False,
            node_size=600, node_color="cyan", font_size=10, font_weight='bold',
            arrows=True)



        plt.text(0.5, 0.95, mensaje, ha='center', va='center', fontsize=12,
            transform=plt.gca().transAxes)

        plt.title('Graph of Node Connections Across Positions')
        plt.axis('off')
        plt.show()

    ############################################################ MAIN
        ##########################################################################################
```

```
337     def main():
338         start_state1 = 1
339         start_state2 = 5
340         target1 = 25
341         target2 = 21
342 
343         all_routes_player1 = "Bloque_1\\Tablero\\all_routes_player1.txt"
344         all_routes_player2 = "Bloque_1\\Tablero\\all_routes_player2.txt"
345         win_routes_player1 = "Bloque_1\\Tablero\\win_routes_player1.txt"
346         win_routes_player2 = "Bloque_1\\Tablero\\win_routes_player2.txt"
347 
348 
349 
350         print("1. Modo autom tico")
351         print("2. Modo manual")
352         opcion = "1"
353         opcion = input("Elige la opci n: ")
354         if opcion == "1":
355             cadena = generar_cadena_rb()
356             print(f"La cadena de movimiento generada es: {cadena}")
357         elif opcion == "2":
358             cadena = str(input("Escribe la cadena de movimiento (ex. rbbbrb): "))
359 
360         player1 = nfa_find_all_paths(cadena, start_state1)
361         player2 = nfa_find_all_paths(cadena, start_state2)
362 
363         write_paths_to_file(player1, all_routes_player1)
364         write_paths_to_file(player2, all_routes_player2)
365         verify_routes(all_routes_player1, win_routes_player1, target1)
366         verify_routes(all_routes_player2, win_routes_player2, target2)
367         print(f"Escribiendo los movimientos '{cadena}' para ambos jugadores.")
368 
369 
370         run_game_animation(win_routes_player1, win_routes_player2)
371         red(all_routes_player1, "Red del jugador 1", target1)
372         red(all_routes_player2, "Red del jugador 2", target2)
373 
374 
375 
376 
377 
378 
379 
380 
381 
382     if __name__ == "__main__":
383         main()
```