INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Reporte de práctica: Tablero con NFA

Grupo:
4CM2

Nombre del alumno:
Maldonado Hernández Alexander

Unidad de aprendizaje:
Teoría de la computación

Maestro:
Genaro Juárez Martínez

Fecha de entrega:
7 de abril de 2025

Índice

1. Introducción

Este reporte documenta el desarrollo de un sistema interactivo y automatizado para la simulación de movimientos de piezas en un tablero de ajedrez reducido de dimensiones 4×4, diseñado como parte de una práctica académica en el marco del estudio de autómatas finitos no deterministas (NFA) y su aplicación en la toma de decisiones. El objetivo principal de la práctica es modelar los movimientos ortogonales y diagonales de dos piezas, cada una controlada por un jugador (autómata) distinto, bajo un conjunto de reglas previamente definidas en las láminas del curso de Stanford.

El sistema debe permitir tanto la operación automática como la manual. En el modo automático, todas las decisiones y movimientos se generan de forma programada, sin intervención del usuario. En cambio, en el modo manual, el usuario tiene la opción de ingresar cadenas de movimientos o bien solicitarlas al sistema de forma aleatoria. Esta dualidad permite analizar el comportamiento del sistema bajo distintos contextos de interacción, lo que resulta valioso para evaluar tanto la robustez como la flexibilidad de la implementación.

Cada jugador cuenta con posiciones específicas de inicio y final: el primer jugador comienza en la casilla 1 (q0) y debe llegar a la casilla 16 (q15), mientras que el segundo jugador inicia en la casilla 4 (q3) y debe llegar a la casilla 13 (q12). El sistema selecciona aleatoriamente cuál jugador inicia la partida, y se encarga de gestionar de forma dinámica los turnos de juego, evitando colisiones y manejando posibles bloqueos en el camino. En caso de rutas imposibles o de conflictos de trayectoria, el sistema implementa una lógica de çeder el paso"que permite continuar con el desarrollo del juego sin interrupciones definitivas.

Además, se estableció un límite máximo de entre 3 y 100 símbolos para la longitud de las cadenas de movimiento, con el fin de mantener un equilibrio entre la complejidad computacional y la viabilidad del análisis.

En conjunto, este proyecto representa una integración efectiva de teoría de autómatas, algoritmos de búsqueda, estructuras de datos, e interfaces gráficas, resultando en una herramienta didáctica útil para la exploración de estrategias en la creación de sistemas autónomos e inteligencia artificial.

2. Desarrollo

El desarrollo del sistema se llevó a cabo mediante un enfoque modular y orientado a objetos, con el objetivo de facilitar la organización del código, su escalabilidad y la implementación de múltiples funcionalidades específicas. El lenguaje de programación utilizado fue Python en su versión 3.12.3, aprovechando bibliotecas como `networkx` para el manejo de grafos, `matplotlib` para la visualización de la red de estados, `tkinter` para la correcta visualización del tablero, `ast` para leer adecuadamente las listas de los archivos de rutas y `random` para la generación de decisiones aleatorias dentro del flujo del programa.

2.1. Estructura general del sistema

El sistema se divide en los siguientes módulos principales:

- **Tablero**: Clase que representa el tablero de 4×4 como un diccionario donde se relaciona cada estado del autómata con una tupla de coordenadas de la forma (x, y).
- **Jugador**: Clase que modela a cada jugador y simula al autómata finito no determinista. Contiene atributos como estado inicial, estado final y identificador.
- **Controlador de Juego**: Administra el flujo general del programa, incluyendo la generación de cadenas de movimiento, la asignación de turnos, el control de la validación de cadenas, así como la detección de condiciones de victoria.
- **Visualización**: Encargado de graficar el tablero y el recorrido de cada jugador sobre el mismo, así como la red de estados del NFA.
- **Archivos de salida**: Módulo que gestiona la escritura de archivos de movimientos posibles y movimientos ganadores por pieza. Además de guardar la gráfica de las rutas del NFA como una imágen con extensión png.

2.2. Modelo de movimiento y reglas

Cada jugador puede desplazarse ortogonalmente (arriba, abajo, izquierda, derecha) o diagonalmente (esquinas) a posiciones válidas dentro del tablero. El sistema valida cada movimiento antes de ejecutarlo. Si el movimiento conduce a una colisión directa con el oponente, se evalúan rutas alternativas para que el jugador no se quede atascado. Si no se encuentra una alternativa válida, el jugador afectado cede el turno. La red de transiciones de cada jugador es modelada como un autómata finito

no determinista (NFA), donde cada estado representa una posición del tablero y cada transición representa un movimiento legal. Se permite la existencia de múltiples caminos hacia el estado final.

2.3. Modos de ejecución

El programa puede ejecutarse en dos modos:

- **Modo Automático**: El sistema genera automáticamente cadenas de movimiento aleatorias para ambos jugadores, simula la partida hasta completarse y genera los archivos correspondientes.
- **Modo Manual**: El usuario puede ingresar manualmente la cadena de movimientos o dejar que el sistema la genere aleatoriamente la cadena para uno de los jugadores o ambos.

2.4. Condiciones iniciales y de finalización

Las condiciones iniciales son fijas y establecidas por los requerimientos de la práctica:

- **Jugador 1:** Comienza en la posición 1 (q0) y debe llegar a la 16 (q15).
- **Jugador 2:** Comienza en la posición 4 (q3) y debe llegar a la 13 (q12).

La finalización del juego ocurre cuando uno o ambos jugadores alcanzan sus posiciones objetivo, o cuando ambos jugadores se quedan atascados en una misma ruta, en cuyo caso se consideraría un empate.

2.5. Generación de archivos de movimiento y graficación

El sistema genera los siguientes archivos:

- **movimientosJX.txt:** Contiene todas las posibles trayectorias de una cadena válida desde el estado inicial hasta cualquier otro estado alcanzable. Se generan dos archivos de este tipo, uno para el jugador uno (J1) y otro para el jugador 2 (J2) con el formato tal cuál como se muestra en la Figura 1.

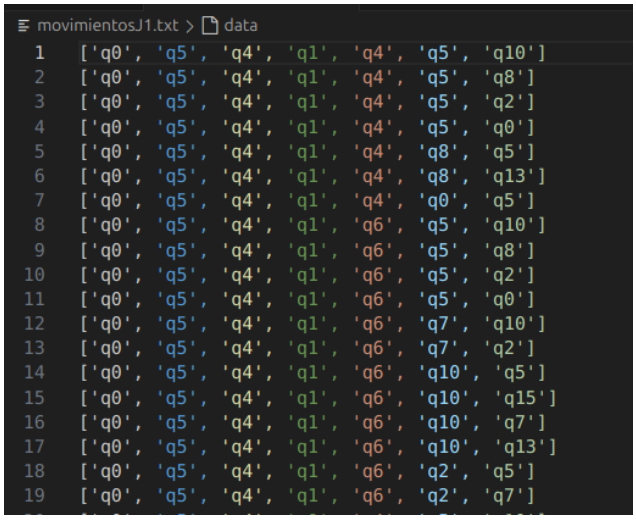


Figura 1: Algunas rutas posibles del jugador 1 con la cadena 'brrrbb'

- **ganadoresJX.txt:** Contiene únicamente las trayectorias que llevan al estado final correspondiente a cada jugador. Se generan dos archivos de este tipo, uno para el jugador uno (J1) y otro para el jugador 2 (J2) con el formato tal cuál como se muestra en la Figura 2.

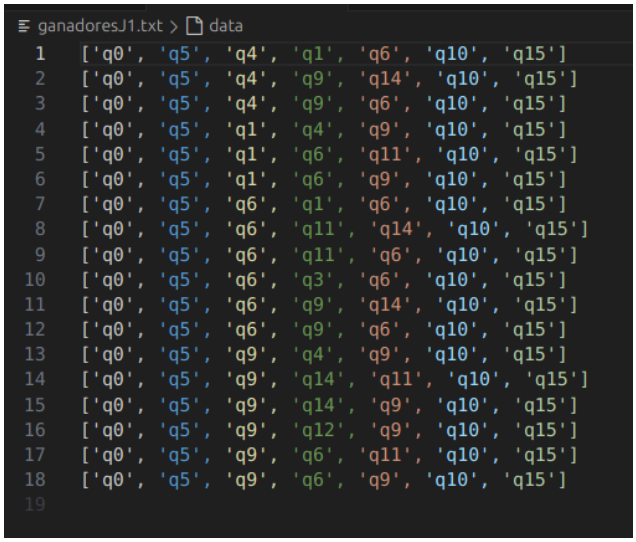


Figura 2: Rutas ganadoras del jugador 1 con la cadena 'brrrbb'

- **GraficaJugadorX.png:** Muestra de manera gráfica todas las posibles rutas que cada jugador podría tomar, sin importar si terminan en el estado final respectivo de cada uno. Se generan dos archivos de este tipo, uno para el jugador uno (Jugador1) y otro para el jugador 2 (Jugador2). Un ejemplo de este archivo se puede ver en la Figura 3

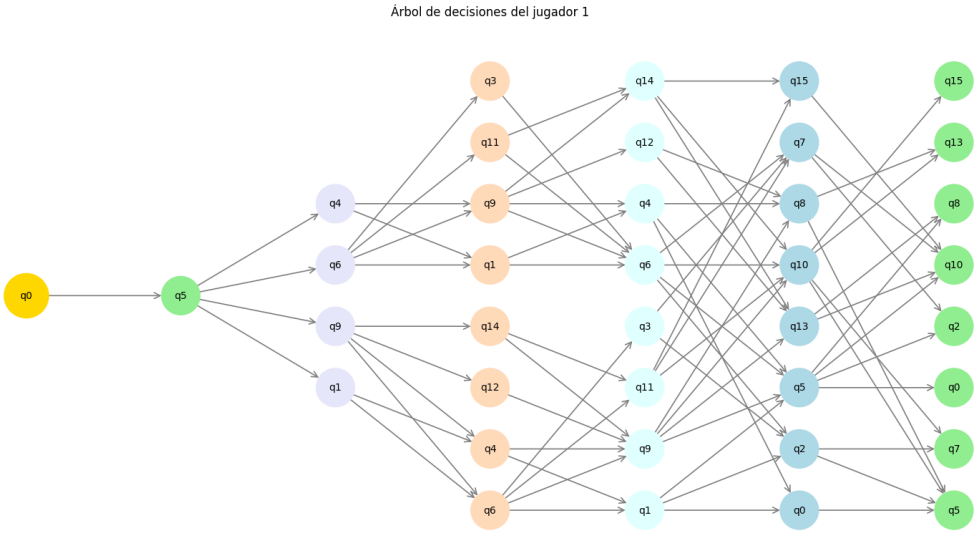


Figura 3: Gráfica de posibles rutas del jugador 1 con la cadena 'brrrbb'

Estos archivos permiten un análisis posterior, así como la validación de rutas viables y estrategias de juego.

2.6. Autómata finito no determinista implementado

En esta práctica, se implementó un autómata finito no determinista (NFA) para modelar los movimientos en el tablero de ajedrez reducido. A continuación, en la Figura 4, se presenta el grafo dirigido que representa al autómata desarrollado, donde cada nodo corresponde a un estado y las transiciones entre ellos son dirigidas, representando los movimientos posibles de las piezas en el tablero.

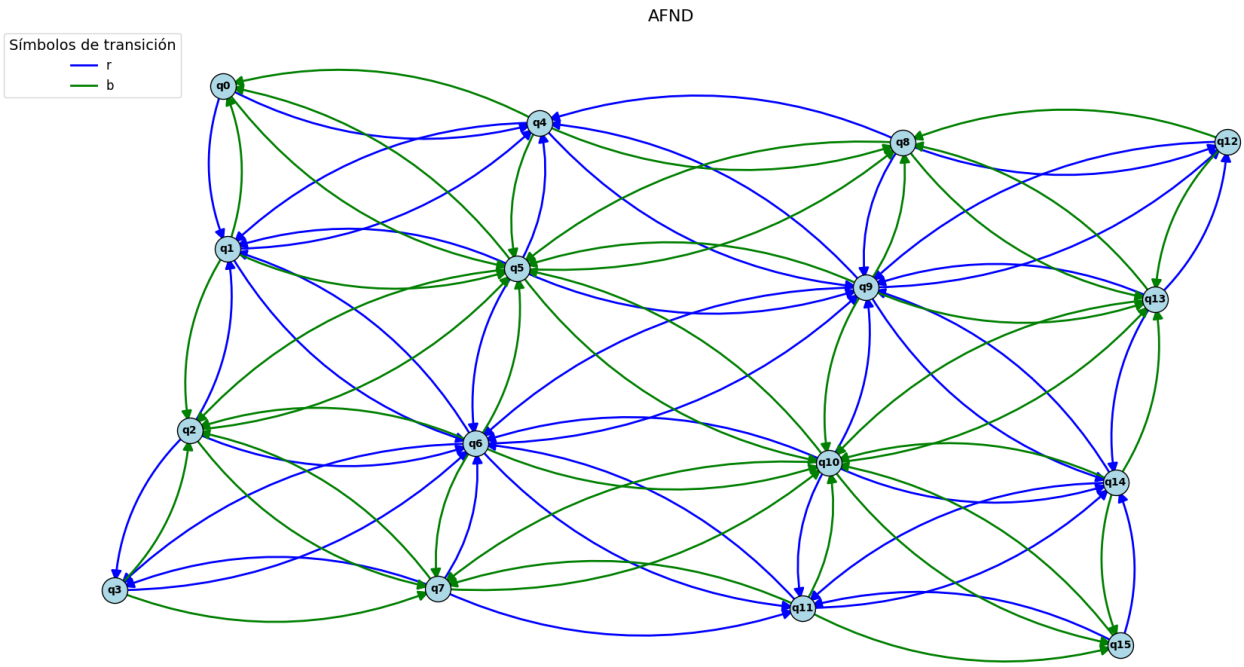
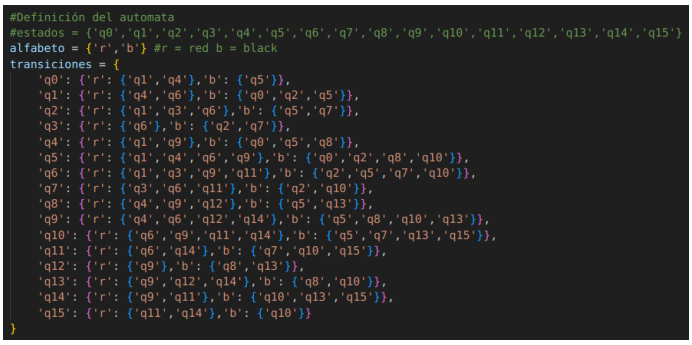


Figura 4: Implementación en código del NFA

Este autómata se caracteriza por ser no determinista, lo que significa que, en ciertos estados, el autómata puede realizar múltiples transiciones hacia otros estados posibles, dependiendo de las decisiones del jugador y las reglas del movimiento en el tablero. Este comportamiento es crucial para modelar los movimientos ortogonales y diagonales de las piezas, ya que, a partir de un estado dado, puede haber varias opciones válidas para el siguiente movimiento.

La implementación de este NFA en el código fuente se llevó a cabo utilizando un diccionario en Python, como se muestra en la Figura 5. Esta estructura de datos fue elegida por su eficiencia en la representación de las transiciones entre estados y su flexibilidad para manejar múltiples posibles transiciones desde un estado dado.



```
#Definición del automata
#estados = {'q0','q1','q2','q3','q4','q5','q6','q7','q8','q9','q10','q11','q12','q13','q14','q15'}
alfabeto = {'r','b'} #r = red b = black
transiciones = {
    'q0': {'r': {'q1','q4'},'b': {'q5'}},
    'q1': {'r': {'q4','q6'},'b': {'q0','q2','q5'}},
    'q2': {'r': {'q1','q3','q6'},'b': {'q5','q7'}},
    'q3': {'r': {'q6'},'b': {'q2','q7'}},
    'q4': {'r': {'q1','q9'},'b': {'q0','q5','q8'}},
    'q5': {'r': {'q1','q4','q6','q9'},'b': {'q0','q2','q8','q10'}},
    'q6': {'r': {'q1','q3','q9','q11'},'b': {'q2','q5','q7','q10'}},
    'q7': {'r': {'q3','q6','q11'},'b': {'q2','q10'}},
    'q8': {'r': {'q4','q9','q12'},'b': {'q5','q13'}},
    'q9': {'r': {'q4','q6','q12','q14'},'b': {'q5','q8','q10','q13'}},
    'q10': {'r': {'q6','q9','q11','q14'},'b': {'q5','q7','q13','q15'}},
    'q11': {'r': {'q6','q14'},'b': {'q7','q10','q15'}},
    'q12': {'r': {'q9'},'b': {'q8','q13'}},
    'q13': {'r': {'q9','q12','q14'},'b': {'q8','q10'}},
    'q14': {'r': {'q9','q11'},'b': {'q10','q13','q15'}},
    'q15': {'r': {'q11','q14'},'b': {'q10'}}
}
```

Figura 5: Implementación en código del NFA

En la implementación, cada clave del diccionario representa un estado dentro del autómata, y los valores asociados a cada clave son listas de transiciones posibles desde ese estado. Cada transición está representada por una combinación de símbolos (movimientos ortogonales o diagonales) que determinan el próximo estado. De esta forma, el diccionario actúa como una tabla de transición de estados, lo que permite consultar rápidamente las rutas disponibles para un jugador en un estado dado.

Por ejemplo, para un estado específico del jugador, el diccionario tiene una entrada con el formato:

```
{ estado: { 'r': { estadoSig1, estadoSig2 }, 'b': { ... } }, ...}
```

Esta estructura facilita la exploración de las posibles rutas dentro del autómata, permitiendo al sistema simular de manera eficiente todos los movimientos posibles que un jugador puede realizar a partir de su posición actual.

La implementación completa de este NFA, así como el código detallado y los ejemplos de su uso, se encuentran en el Anexo 1 de este reporte. Este anexo proporciona una visión más profunda sobre cómo se organizan las transiciones en el autómata y cómo interactúan con las demás partes del sistema, como el controlador de juego y la visualización gráfica del tablero.

3. Conclusiones

El desarrollo de este proyecto representó una oportunidad significativa para aplicar conocimientos teóricos de los autómatas finitos no deterministas (NFA) a un entorno práctico y dinámico. A través de la simulación de movimientos ortogonales y diagonales en un tablero de ajedrez reducido de 4×4, se logró construir un sistema capaz de gestionar la lógica compleja de interacción entre dos jugadores, considerando no solo las trayectorias individuales, sino también la influencia mutua de sus decisiones y movimientos sobre el tablero compartido.

La distinción entre los modos automático y manual aportó versatilidad al sistema. El modo automático permitió validar el comportamiento global, mientras que el modo manual facilitó la exploración detallada de casos particulares. Esta característica fue clave para garantizar la integridad de las trayectorias generadas y la correcta implementación de las reglas establecidas.

La visualización del tablero y de la red NFA generada brindó una ventaja significativa en términos de interpretación e interpretación del comportamiento del sistema. La representación gráfica de los nodos, transiciones, rutas activas y trayectorias ganadoras contribuyó a una comprensión más clara del flujo del juego y facilitó la identificación de patrones, colisiones o cuellos de botella que podrían surgir durante el transcurso de una partida.

Entre los desafíos más relevantes se encontraron el manejo de bloqueos, la definición de reglas de "cedencia de turno" y la validación de rutas viables en presencia de múltiples posibilidades de movimiento. Resolver estas situaciones exigió un análisis detallado del estado global del sistema y la implementación de estrategias que simulen comportamientos inteligentes, sin salirse de los límites definidos por los autómatas. En este sentido, se logró construir un motor de decisiones que, aunque limitado por la no determinación, fue capaz de adaptarse a una amplia gama de escenarios.

Este proyecto me permitió reforzar conceptos fundamentales de la teoría de autómatas, grafos y algoritmos de búsqueda. El resultado fue un sistema funcional, visualmente interactivo y académicamente enriquecedor, que puede servir como base para proyectos más ambiciosos en el campo de la simulación de sistemas inteligentes, juegos estratégicos o la computación autónoma.

4. Referencias

FAMAF. (s.f.). Automatas finitos no deterministas. https://cs.famaf.unc.edu.ar/~hoffmann/md20/08.html#/title-slide

Foundation, P. S. (2025, abril). Graphical User Interfaces with Tk. https://docs.python.org/3/library/tk.html

Hunter, J., Dale, D., Firing, E., & Droettboom, M. (2025). Matplotlib 3.10.1 documentation. https://matplotlib.org/stable/index.html

NetworkX. (2024, octubre). NetworkX documentation. https://networkx.org/

5. Anexos

5.1. Anexo 1. Código fuente del programa

```
import random as rd
import tkinter as tk
from tkinter import messagebox
import ast
import networkx as nx
import matplotlib.pyplot as plt
from collections import defaultdict

class Graficador:
    def __init__(self, archivo, jugador):
        self.archivo = archivo
        self.jugador = jugador
        self.leer_rutas()
        self.construir_arbol()
        self.calcular_profundidad()
        self.definir_posiciones()
        self.dibujar_arbol()

    def leer_rutas(self):
        with open(self.archivo, 'r') as f:
            self.rutas = [ast.literal_eval(linea.strip()) for linea
            ↪  in f if linea.strip()]

    def construir_arbol(self):
        #Crea el árbol y asigna un identificador único a cada nodo
        ↪  basado en su posición.
        self.arbol = nx.DiGraph()
        self.nodos_info = {}
        rutas_ids = []

        for ruta in self.rutas:
            identificadores = []
            for pos, estado in enumerate(ruta):
                nodo_id = f"{estado}_{pos}_{len(identificadores)}"
```

```
                identificadores.append(nodo_id)
                self.nodos_info[nodo_id] = {'estado': estado,
                ↪  'profundidad': pos}
            rutas_ids.append(identificadores)

        #Se añaden las aristas según la secuencia de cada ruta.
        for ids in rutas_ids:
            for i in range(len(ids) - 1):
                self.arbol.add_edge(ids[i], ids[i + 1])

    def calcular_profundidad(self):
        #Calcula la profundidad de cada nodo a partir de la
        ↪  información previamente guardada.
        self.profundidad = {nodo: info['profundidad'] for nodo, info
        ↪  in self.nodos_info.items()}

    def definir_posiciones(self):
        #Define las posiciones para la visualización del árbol
        ↪  usando un espaciado vertical uniforme.
        self.posiciones = {}
        nodos_por_nivel = defaultdict(list)

        for nodo, nivel in self.profundidad.items():
            nodos_por_nivel[nivel].append(nodo)

        separacion_y = 1.5  # Espaciado vertical entre nodos

        for nivel, nodos in nodos_por_nivel.items():
            total = len(nodos)
            for idx, nodo in enumerate(nodos):
                x = nivel  # La posición horizontal representa la
                ↪  profundidad
                # Distribución centrada en el eje vertical
                y = (idx - (total - 1) / 2) * separacion_y if total >
                ↪  1 else 0
                self.posiciones[nodo] = (x, y)

    def dibujar_arbol(self):
```

```
        fig, ax = plt.subplots(figsize=(16, 8))

        # Se prepara un diccionario de etiquetas para mostrar solo
        ↪  el estado de cada nodo.
        etiquetas = {nodo: self.nodos_info[nodo]['estado'] for nodo
        ↪  in self.arbol.nodes()}

        # Configuración de colores según la profundidad del nodo.
        paleta = ['lightblue', 'lightgreen', 'lavender', 'peachpuff',
        ↪  'lightcyan']
        colores_nodos = [paleta[self.profundidad[nodo] % len(paleta)]
        ↪  for nodo in self.arbol.nodes()]

        # Dibuja el árbol con las etiquetas y colores definidos.
        nx.draw(self.arbol, self.posiciones, ax=ax, with_labels=True,
        ↪  labels=etiquetas, node_size=1500,
        ↪  node_color=colores_nodos, edge_color="gray",
        ↪  font_size=10, arrowsize=15, arrowstyle='->', width=1.2)

        #Resalta el nodo de la raíz (profundidad 0) con un color
        ↪  distinto.
        raiz = [nodo for nodo in self.arbol.nodes() if
        ↪  self.profundidad[nodo] == 0]
        if raiz:
            nx.draw_networkx_nodes(self.arbol, self.posiciones,
            ↪  nodelist=raiz, node_size=2000, node_color="gold")

        plt.title(f"Árbol de decisiones del jugador {self.jugador}",
        ↪  pad=20)
        plt.tight_layout()
        plt.savefig(f"GraficaJugador{self.jugador}", dpi=300)
        plt.show()

class Jugador:
    def __init__(self, transiciones, estado_inicial, estado_final,
    ↪  idArchivo):
        self.transiciones = transiciones
        self.estado_inicial = estado_inicial
```

```
        self.estado_final = estado_final
        self.idArchivo = idArchivo

        self.limpiar_archivos()

    def limpiar_archivos(self):
        open(f"movimientos{self.idArchivo}.txt","w").close()
        open(f"ganadores{self.idArchivo}.txt","w").close()

    #Checa si una cadena es ganadora
    def es_ganadora(self,ruta):
        if ruta[-1] == self.estado_final:
            return True
        else:
            return False

    #Obtiene todas las rutas posibles del automata
    def explorar_rutas(self, cadena, estado_actual = None,
    ↪  ruta_actual = None):
        if ruta_actual is None or estado_actual is None:
            ruta_actual = []
            estado_actual = self.estado_inicial

        ruta_actual = ruta_actual + [estado_actual]

        if len(cadena) == 0:
            with open(f"movimientos{self.idArchivo}.txt","a") as m,
            ↪  open(f"ganadores{self.idArchivo}.txt","a") as g:
                if self.es_ganadora(ruta_actual) == True:
                    g.write(f"{ruta_actual}\n")
                    m.write(f"{ruta_actual}\n")
                    return
                else:
                    m.write(f"{ruta_actual}\n")
                    return
        else:
            for estado in
            ↪  self.transiciones[estado_actual][cadena[0]]:
```

```
                self.explorar_rutas(cadena[1:],estado,ruta_actual)

    #Se obtienen las rutas gandoras calculadas anteriormente
    def get_rutas(self):
        with open(f"ganadores{self.idArchivo}.txt","r") as g:
            rutas_ganadoras = []
            for ruta in g:

                ↪  rutas_ganadoras.append(ast.literal_eval(ruta.strip()))
        return rutas_ganadoras

    #Se asegura que haya al menos una ruta ganadora para la cadena
    def es_cadena_valida(self):
        with open(f"ganadores{self.idArchivo}.txt","r") as g:
            ruta = g.readline()
            if not ruta:
                return False
            else:
                return True


class Tablero:
    def __init__(self, ventana, rutasJ1, rutasJ2):
        self.ventana = ventana
        self.ventana.title("Tablero")
        self.rutasJ1 = rutasJ1
        self.rutasJ2 = rutasJ2
        self.reconf1 = False
        self.reconf2 = False
        self.cedido1 = 0
        self.cedido2 = 0


        #Mapeo de estados a posiciones en el tablero (fila, columna)
        self.posiciones = {
            'q0': (0, 0), 'q1': (0, 1), 'q2': (0, 2), 'q3': (0, 3),
            'q4': (1, 0), 'q5': (1, 1), 'q6': (1, 2), 'q7': (1, 3),
            'q8': (2, 0), 'q9': (2, 1), 'q10': (2, 2), 'q11': (2, 3),
```

```
            'q12': (3, 0), 'q13': (3, 1), 'q14': (3, 2), 'q15': (3,
            ↪  3)
        }

        self.colores = {
            'negro': "#000000",
            'rojo': "#FF0000"
        }

        self.crearTablero()
        self.crearJugadores()
        self.iniciarJuego()

    def crearTablero(self):
        for fila in range(4):
            for columna in range(4):
                color = self.colores['negro'] if (fila + columna) % 2
                ↪  == 0 else self.colores['rojo']
                casilla = tk.Label(self.ventana, bg=color, width=13,
                ↪  height=5, relief="ridge")
                casilla.grid(row=fila, column=columna)

    def crearJugadores(self):
        self.pieza1 = tk.Canvas(self.ventana, width=70, height=70,
        ↪  bg="white",highlightthickness=0)
        self.pieza1.grid(row=0,column=0)
        self.pieza2 = tk.Canvas(self.ventana, width=70, height=70,
        ↪  bg="yellow",highlightthickness=0)
        self.pieza2.grid(row=0,column=3)

    def iniciarJuego(self):
        self.rutaJ1 = list(rd.choice(self.rutasJ1)).copy()
        self.rutaJ2 = list(rd.choice(self.rutasJ2)).copy()

        print(f"Ruta a jugar para el jugador 1: {self.rutaJ1}\nRuta a
        ↪  jugar para el jugador 2: {self.rutaJ2}")

        self.estado_actual1 = self.rutaJ1.pop(0)
```

```
        self.estado_actual2 = self.rutaJ2.pop(0)

        self.turno = rd.choice(['1','2'])
        print(f"El jugador {self.turno} comenzará la partida")

        self.ventana.after(1000,self.jugarTurno)

    def reconfigurarRuta(self):
        subrutas = []
        if self.turno == '1':
            restante = len(self.rutasJ1[0]) - len(self.rutaJ1) - 1
            ↪ #Se le resta un uno para tomar en cuenta el estado
            ↪ actual
            for ruta in self.rutasJ1:
                ruta = ruta[restante:]
                if ruta[0] == self.estado_actual1 and ruta[1] !=
                ↪ self.estado_actual2:
                    subrutas.append(ruta[1:])
            return rd.choice(subrutas) if subrutas else self.rutaJ2
            ↪ #En caso de no encontrar ninguna ruta vuelve a usar
            ↪ la actual
        else:
            restante = len(self.rutasJ2[0]) - len(self.rutaJ2) - 1
            for ruta in self.rutasJ2:
                ruta = ruta[restante:]
                if ruta[0] == self.estado_actual2 and ruta[1] !=
                ↪ self.estado_actual1:
                    subrutas.append(ruta[1:])
            return rd.choice(subrutas) if subrutas else self.rutaJ2

    def jugarTurno(self):
        #Verifica condiciones de victoria
        if not self.rutaJ1 or not self.rutaJ2:
            if self.pieza1.grid_info()['row'] == 3 and
            ↪ self.pieza1.grid_info()['column'] == 3:
                messagebox.showinfo(title="Terminó el
                ↪ juego",message="¡El jugador 1 ha ganado!")
                self.ventana.destroy()
```

```
            return
        elif self.pieza2.grid_info()['row'] == 3 and
        ↪  self.pieza2.grid_info()['column'] == 0:
            messagebox.showinfo(title="Terminó el
            ↪  juego",message="¡El jugador 2 ha ganado!")
            self.ventana.destroy()
            return

    while True:
        #Verifica a quien le toca su turno o si los jugadores
        ↪  estan atascados
        if self.cedido1 > 2 or self.cedido2 > 2:
            messagebox.showinfo(title="Terminó el
            ↪  juego",message="Los jugadores han quedado
            ↪  empatados")
            self.ventana.destroy()
            return
        elif self.turno == '1':
            estado_sig1 = self.rutaJ1[0]
            if estado_sig1 == self.estado_actual2 and
            ↪  self.reconf1 == True: #Si ya se ha reconfigurado
            ↪  y sigue atascado se cede el turno
                print("\nEl jugador 1 ha cedido su turno")
                self.reconf1 = False
                self.cedido1 += 1
                break
            elif estado_sig1 == self.estado_actual2:
            ↪  #Verifica si la casilla siguiente esta ocupada,
            ↪  si lo esta, reconfigura la ruta
                self.rutaJ1 = self.reconfigurarRuta()
                self.reconf1 = True
                print(f"\nSe ha reconfigurado la ruta restante
                ↪  del jugador 1 a {[self.estado_actual1] +
                ↪  self.rutaJ1}")
                continue
            else:                                              #Se
            ↪  mueve el jugador si la casilla esta desocupada o
            ↪  se ha reconfigurado a una ruta adecuada
```

```
                    print(f"\nJugador 1: {self.estado_actual1} ==>
                    ↪ {estado_sig1}")
                    self.estado_actual1 = self.rutaJ1.pop(0)

                    ↪ self.pieza1.grid(row=self.posiciones[self.estado_actual1][0],
                    self.reconf1 = False
                    self.cedido1 = 0
                    break
            else:
                estado_sig2 = self.rutaJ2[0]
                if estado_sig2 == self.estado_actual1 and
                ↪ self.reconf2 == True:
                    print("\nEl jugador 2 ha cedido su turno")
                    self.reconf2 = False
                    self.cedido2 += 1
                    break
                elif estado_sig2 == self.estado_actual1:
                    self.rutaJ2 = self.reconfigurarRuta()
                    self.reconf2 = True
                    print(f"\nSe ha reconfigurado la ruta restante
                    ↪ del jugador 2 a {[self.estado_actual2] +
                    ↪ self.rutaJ2}")
                    continue
                else:
                    print(f"\nJugador 2: {self.estado_actual2} ==>
                    ↪ {estado_sig2}")
                    self.estado_actual2 = self.rutaJ2.pop(0)

                    ↪ self.pieza2.grid(row=self.posiciones[self.estado_actual2][0],
                    self.reconf2 = False
                    self.cedido2 = 0
                    break

        #Alternar turno y repetir después de 1000 ms
        self.turno = '2' if self.turno == '1' else '1'
        self.ventana.after(1000, self.jugarTurno)

def generar_cadena(jugador,tamano):
```

```
    #jugador = 1 para referirse al jugador 1, jugador = 2 para
    ↪  refererise al jugador 2
    if tamano == 2:
        return "bbb" if jugador == 1 else "rrr" #Unica cadena valida
        ↪  de 3 movimientos
    else:
        return "".join(rd.choices(population='rb', k=tamano)) + ('b'
        ↪  if jugador == 1 else 'r')

def main():

    #Definición del automata
    #estados =
    ↪  {'q0','q1','q2','q3','q4','q5','q6','q7','q8','q9','q10','q11','q12','q13',
    alfabeto = {'r','b'} #r = red b = black
    transiciones = {
        'q0': {'r': {'q1','q4'},'b': {'q5'}},
        'q1': {'r': {'q4','q6'},'b': {'q0','q2','q5'}},
        'q2': {'r': {'q1','q3','q6'},'b': {'q5','q7'}},
        'q3': {'r': {'q6'},'b': {'q2','q7'}},
        'q4': {'r': {'q1','q9'},'b': {'q0','q5','q8'}},
        'q5': {'r': {'q1','q4','q6','q9'},'b':
        ↪  {'q0','q2','q8','q10'}},
        'q6': {'r': {'q1','q3','q9','q11'},'b':
        ↪  {'q2','q5','q7','q10'}},
        'q7': {'r': {'q3','q6','q11'},'b': {'q2','q10'}},
        'q8': {'r': {'q4','q9','q12'},'b': {'q5','q13'}},
        'q9': {'r': {'q4','q6','q12','q14'},'b':
        ↪  {'q5','q8','q10','q13'}},
        'q10': {'r': {'q6','q9','q11','q14'},'b':
        ↪  {'q5','q7','q13','q15'}},
        'q11': {'r': {'q6','q14'},'b': {'q7','q10','q15'}},
        'q12': {'r': {'q9'},'b': {'q8','q13'}},
        'q13': {'r': {'q9','q12','q14'},'b': {'q8','q10'}},
        'q14': {'r': {'q9','q11'},'b': {'q10','q13','q15'}},
        'q15': {'r': {'q11','q14'},'b': {'q10'}}
    }
```

```
#Instanciando a los jugadores
jugador1 = Jugador(transiciones,'q0','q15','J1')
jugador2 = Jugador(transiciones,'q3','q12','J2')


while True:
    entrada = input("1. Modo manual\n2. Modo automático\n3.
    ↪  Salir\n").strip()

    if entrada not in ['1','2','3']:
        print("Entrada inválida. Introduzca '1', '2' o '3'")
        continue
    else:
        if entrada == '1':
            while True:
                cadena1 = input("\nInserte la cadena de símbolos
                ↪  'r' y 'b' para el jugador 1 (3 a 100
                ↪  símbolos) o presione Enter para generar una
                ↪  cadena aleatoria: \n").lower().replace("
                ↪  ","")

                if cadena1 == "":
                    while True:
                        cadena1 = generar_cadena(1,
                        ↪  rd.randint(2,99))
                        print(f"Calculando rutas con la cadena
                        ↪  '{cadena1}' para el jugador 1...")
                        jugador1.explorar_rutas(cadena1)

                        if not jugador1.es_cadena_valida():
                            print("No se encontraron rutas
                            ↪  ganadoras para el jugador 1.
                            ↪  Generando nueva cadena...\n")
                            jugador1.limpiar_archivos()
                            continue
                        else:
                            print("Rutas del jugador 1 calculadas
                            ↪  exitosamente\n")
```

```
                break
            break

        elif cadena1[-1] != 'b' or not set(cadena1) <=
        ↪  alfabeto or not 3 <= len(cadena1) <= 100:
            print("Cadena inválida. Solo se aceptan los
            ↪  caracteres 'b' y 'r' en una cadena de 3 a
            ↪  100 caracteres, además la cadena debe de
            ↪  terminar con una 'b'")
            continue

        else:
            print("Cadena aceptada. Calculando rutas...")
            jugador1.explorar_rutas(cadena1)
            if not jugador1.es_cadena_valida():
                print("No se encontraron rutas ganadoras
                ↪  para el jugador 1. Pruebe con otra
                ↪  cadena\n")
                jugador1.limpiar_archivos()
                continue
            else:
                print("Rutas del jugador 1 calculadas
                ↪  exitosamente\n")
                break

while True:
    cadena2 = input("\nInserte la cadena de símbolos
    ↪  'r' y 'b' para el jugador 2 (3 a 100
    ↪  símbolos) o presione Enter para generar una
    ↪  cadena aleatoria: \n").lower().replace("
    ↪  ","")

    if cadena2 == "":
        while True:
            cadena2 = generar_cadena(2, len(cadena1)
            ↪  - 1)
            print(f"Calculando rutas con la cadena
            ↪  '{cadena2}' para el jugador 2...")
```

```
                jugador2.explorar_rutas(cadena2)

                if not jugador2.es_cadena_valida():
                    print("No se encontraron rutas
                    ↪  ganadoras para el jugador 2.
                    ↪  Generando nueva cadena...\n")
                    jugador2.limpiar_archivos()
                    continue
                else:
                    print("Rutas del jugador 2 calculadas
                    ↪  exitosamente\n")
                    break
            break

        elif cadena2[-1] != 'r' or not set(cadena2) <=
        ↪  alfabeto or len(cadena2) != len(cadena1):
            print("Cadena inválida. Solo se aceptan los
            ↪  caracteres 'b' y 'r' en una cadena de
            ↪  longitud igual a la anterior, además la
            ↪  cadena debe de terminar con una 'r'")
            continue

        else:
            print("Cadena aceptada. Calculando rutas...")
            jugador2.explorar_rutas(cadena2)
            if not jugador2.es_cadena_valida():
                print("No se encontraron rutas ganadoras
                ↪  para el jugador 2. Pruebe con otra
                ↪  cadena\n")
                jugador1.limpiar_archivos()
                continue
            else:
                print("Rutas del jugador 2 calculadas
                ↪  exitosamente\n")
                break

print("Rutas calculadas. Comenzando juego de
↪  ajedrez...")
```

```
elif entrada == '2':
    n = rd.randint(2, 99)

    while True:
        cadena1 = generar_cadena(1, n)
        print(f"Calculando rutas con la cadena
        ↪  '{cadena1}' para el jugador 1...")
        jugador1.explorar_rutas(cadena1)

        if not jugador1.es_cadena_valida():
            print("No se encontraron rutas ganadoras para
            ↪  el jugador 1. Generando nueva
            ↪  cadena...\n")
            jugador1.limpiar_archivos()
            continue
        else:
            print("Rutas del jugador 1 calculadas
            ↪  exitosamente\n")
            break

    while True:
        cadena2 = generar_cadena(2, n)
        print(f"Calculando rutas con la cadena
        ↪  '{cadena2}' para el jugador 2...")
        jugador2.explorar_rutas(cadena2)

        if not jugador2.es_cadena_valida():
            print("No se encontraron rutas ganadoras para
            ↪  el jugador 2. Generando nueva
            ↪  cadena...\n")
            jugador2.limpiar_archivos()
            continue
        else:
            print("Rutas del jugador 2 calculadas
            ↪  exitosamente\n")
            break
```

```
            print("Comenzando juego de ajedrez...\n")

        elif entrada == '3':
            break

        #Se muestran los arboles de rutas de cada jugador
        Graficador("movimientosJ1.txt","1")
        Graficador("movimientosJ2.txt","2")

        #Se inicia el juego
        ventana = tk.Tk()
        Tablero(ventana, jugador1.get_rutas(),
        ↪  jugador2.get_rutas())
        ventana.mainloop()

        jugador1.limpiar_archivos()
        jugador2.limpiar_archivos()

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