Tablero Ajedrez 5x5

Estudiante González Fonseca José Juan
Docente Juarez Martinez Genaro
Materia Teoria de la Computación

Introducción

En esta práctica realizamos la codificación necesaria para un autómata finito no determinístico (NFA) para mover un rey sobre un tablero de 5x5 desde una posición inicial hasta una posición deseada, enumerados del 1 al 25, donde cada casilla es un estado diferente.
Como estado inicial para el rey blanco, tenemos la posición 1 y un estado final en la posición 25, mientras que el rey negro empieza en el estado 5 y termina en el estado 21, todo esto definido sobre el alfabeto $\Sigma = \{'r', 'b'\}$.
Adicionalmente, mostraremos gráficamente una ruta ganadora para cada rey con la capacidad de reconfigurar sus rutas para evitar el sobreposicionamiento de las piezas. Además, generaremos las gráficas de las redes creadas por todos los movimientos de ambos jugadores.

Marco Teorico

Autómatas Finitos No Determinísticos (NFA)

Un autómata finito no determinístico (NFA, por sus siglas en inglés) es un modelo matemático de computación que se utiliza para representar sistemas con múltiples estados y transiciones posibles. Un NFA se define formalmente por un conjunto de estados, un conjunto de símbolos de entrada (alfabeto), una función de transición que permite múltiples posibles movimientos, un estado inicial y un conjunto de estados de aceptación.
El comportamiento no determinístico permite que, dado un estado actual y un símbolo de entrada, el autómata pueda "elegir" entre varias posibles transiciones. En esta práctica, el NFA modela los posibles movimientos de los reyes en el tablero de 5x5, donde cada casilla corresponde a un estado diferente y el alfabeto representa los turnos de cada jugador.

Grafos y Redes de Estados

Los grafos son estructuras matemáticas que se utilizan para modelar relaciones entre objetos. En un grafo, los nodos (o vértices) representan los estados, mientras que las aristas (o conexiones) representan las transiciones entre estos estados.
En esta práctica, el tablero de 5x5 puede representarse como un grafo donde cada casilla es un nodo y las transiciones posibles (movimientos del rey) son las aristas que conectan estos nodos. La visualización gráfica de los movimientos de los reyes se corresponde con un recorrido sobre este grafo.

Movimientos del Rey en Ajedrez

En ajedrez, el rey es una pieza que puede moverse una casilla en cualquier dirección: horizontal, vertical o diagonal. En esta práctica, simulamos estos movimientos restringidos en un tablero de 5x5.
El objetivo de cada rey es llegar a su casilla final, evitando sobreponerse con el otro rey. Los movimientos permitidos están restringidos por las reglas de movimiento del rey en ajedrez y limitados a la cadena ingresada.

Desarrollo

Para esta práctica usamos 6 programas hechos en Python. Empezamos por el programa principal desde donde se llamarán a los otros 5.

- Empezamos pidiendo al usuario si quiere correr en modo manual o automático.
- En el modo manual, el usuario deberá ingresar las cadenas para el rey blanco y el rey negro, cadenas que seguirán para buscar todas las rutas.

- En el modo automático, el programa creará automáticamente estas cadenas con un máximo de 100 caracteres.
- En ambos casos, se procederá a ejecutar el cálculo de las rutas, la representación del tablero y la gráfica de ambas redes llamando a los programas necesarios.
- Todo funciona en un bloque que se repite hasta que el usuario desee salir.

Listing 1: Programa principal

```
import subprocess
import random

def ejecutar_programa(programa, args):
    subprocess.run(['python', programa] + args)

def modo_manual():
    """Ejecuta en modo manual pidiendo al usuario las cadenas."""
    while True:
        cadena_blanco = input("Ingresa la cadena para el rey blanco (solo 'r' y 'b'): ")
        cadena_negro = input("Ingresa la cadena para el rey negro (solo 'r' y 'b'): ")

        # Contar los caracteres en cadena_blanco
        contador_blanco = 0
        for _ in cadena_blanco:
            contador_blanco += 1

        # Contar los caracteres en cadena_negro
        contador_negro = 0
        for _ in cadena_negro:
            contador_negro += 1

        if contador_blanco != contador_negro:
            print("Las cadenas deben ser de la misma longitud. Intenta de nuevo.")
            continue

        if not all(c in 'rb' for c in cadena_blanco + cadena_negro):
            print("Las cadenas solo pueden contener 'r' y 'b'. Intenta de nuevo.")
            continue

        # Ejecutar los programas pasando las cadenas como argumento
        ejecutar_programa('rutaRB.py', [cadena_blanco])
        ejecutar_programa('rutaRN.py', [cadena_negro])

        ejecutar_programa('tablero.py', [])
        ejecutar_programa('redRB.py', [])
        ejecutar_programa('redRN.py', [])

        # Preguntar si desea continuar
        continuar = input("Quieres continuar? (s/n): ").lower()
        if continuar != 's':
            break


def modo_automatico():
    """Ejecuta en modo automatico generando cadenas aleatorias."""
    while True:
        longitud = random.randint(1, 100)
        cadena_blanco = ''.join(random.choice('rb') for _ in range(longitud))
        cadena_negro = ''.join(random.choice('rb') for _ in range(longitud))

        print(f"Cadenas generadas: Blanco = {cadena_blanco}, Negro = {cadena_negro}")

        # Ejecutar los programas pasando las cadenas como argumento
```

```
        ejecutar_programa('rutaRB.py', [cadena_blanco])
        ejecutar_programa('rutaRN.py', [cadena_negro])

        ejecutar_programa('tablero.py', [])

        ejecutar_programa('redRB.py', [])
        ejecutar_programa('redRN.py',[])


        # Preguntar si desea continuar
        continuar = input("Quieres continuar? (s/n): ").lower()
        if continuar != 's':
            break

def main():
    while True:
        print("Seleccione un modo:")
        print("1. Modo Manual")
        print("2. Modo Automatico")
        print("3. Salir")

        eleccion = input("Ingrese su eleccion (1, 2, o 3): ")

        if eleccion == '1':
            modo_manual()
        elif eleccion == '2':
            modo_automatico()
        elif eleccion == '3':
            break
        else:
            print("Opcion no valida. Intenta de nuevo.")

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

Los siguientes dos programas son para la generación de todos los movimientos del rey blanco y el rey negro.

- Empezamos recibiendo como argumento la cadena para generar los movimientos, esta cadena es recibida desde el programa principal.
- Validamos una vez más que la cadena ingresada esté definida en nuestro alfabeto.
- Definimos un diccionario de cada casilla (estado) con las posiciones adyacentes que tiene y los colores de esta, esto servirá para saber en qué estado puede continuar nuestra ruta dependiendo de la cadena de entrada.
- Generamos una lista donde iremos guardando todos los movimientos.
- Usamos una función recursiva. En el caso base, si la secuencia restante está vacía, guardaremos este movimiento en nuestra lista. Para los demás casos, identificaremos el color del siguiente movimiento y buscaremos las posiciones adyacentes posibles gracias al diccionario que creamos previamente; desde aquí llamamos recursivamente.
- Por último, guardamos las rutas en tres archivos `txt`: uno para todas las rutas, otro para las rutas ganadoras, y un último para las rutas perdedoras de cada rey.
- La única diferencia entre un programa y otro es la posición inicial y la posición final dependiendo del rey.

Listing 2: Programa para el calculo de rutas del rey blanco

```
import sys

def generar_movimientos(tablero, secuencia, posicion_inicial=1):
    movimientos = []

    # Definir el color que corresponde a 'r' y 'b'
    colores = {'r': 'rojo', 'b': 'negro'}
```

```
8
9       # Funcion recursiva para generar las combinaciones de movimientos
10      def explorar(posicion_actual, secuencia_restante, ruta):
11          if not secuencia_restante:  # Si la secuencia esta vacia, guardar la ruta
12              movimientos.append(ruta)
13              return
14
15          color_deseado = secuencia_restante[0]  # Color que queremos respetar en este
                paso
16          adyacentes = tablero[posicion_actual]  # Posiciones adyacentes de la actual
17
18          # Verifica si el color deseado esta en el diccionario
19          if color_deseado not in adyacentes:
20              return  # Si el color no esta, terminar la busqueda
21
22          # Explorar cada posicion adyacente
23          for adyacente in adyacentes[color_deseado]:
24              # Si el color coincide, explorar mas desde esta posicion
25              explorar(adyacente, secuencia_restante[1:], ruta + [adyacente])
26
27      # Iniciar la busqueda desde la posicion inicial
28      explorar(posicion_inicial, secuencia, [posicion_inicial])
29
30      return movimientos
31
32
33  # Definir posiciones adyacentes (por color) para el tablero de ajedrez de 5x5
34  tablero = {
35      1: {'r': [2, 6], 'b': [7]},
36      2: {'r': [6, 8], 'b': [1, 3, 7]},
37      3: {'r': [2, 4, 8], 'b': [7, 9]},
38      4: {'r': [8, 10], 'b': [3, 5, 9]},
39      5: {'r': [4, 10], 'b': [9]},
40      6: {'r': [2, 12], 'b': [1, 7, 11]},
41      7: {'r': [2, 6, 8, 12], 'b': [1, 3, 11, 13]},
42      8: {'r': [2, 4, 12, 14], 'b': [3, 7, 9, 13]},
43      9: {'r': [4, 8, 10, 14], 'b': [3, 5, 13, 15]},
44      10: {'r': [4, 14], 'b': [5, 9, 15]},
45      11: {'r': [6, 12, 16], 'b': [7, 17]},
46      12: {'r': [6, 8, 16, 18], 'b': [7, 11, 13, 17]},
47      13: {'r': [8, 12, 14, 18], 'b': [7, 9, 17, 19]},
48      14: {'r': [8, 10, 18, 20], 'b': [9, 13, 15, 19]},
49      15: {'r': [10, 14, 20], 'b': [9, 19]},
50      16: {'r': [12, 22], 'b': [11, 17, 21]},
51      17: {'r': [12, 16, 18, 22], 'b': [11, 13, 21, 23]},
52      18: {'r': [12, 14, 22, 24], 'b': [13, 17, 19, 23]},
53      19: {'r': [14, 18, 20, 24], 'b': [13, 15, 23, 25]},
54      20: {'r': [14, 24], 'b': [15, 19, 25]},
55      21: {'r': [16, 22], 'b': [17]},
56      22: {'r': [16, 18], 'b': [17, 21, 23]},
57      23: {'r': [18, 22, 24], 'b': [17, 19]},
58      24: {'r': [18, 20], 'b': [19, 23, 25]},
59      25: {'r': [20, 24], 'b': [19]}
60  }
61
62  # Funcion para guardar las rutas en los archivos correspondientes
63  def guardar_rutas(movimientos):
64      with open("Rutas/todas_las_rutasRB.txt", "w") as todas, open("Rutas/
            rutas_ganadorasRB.txt", "w") as ganadoras, open("Rutas/rutas_perdedorasRB.txt
            ", "w") as perdedoras:
65          for movimiento in movimientos:
66              ruta_str = "␣->␣".join(map(str, movimiento))  # Convertir la ruta en una
                    cadena legible
67              todas.write(ruta_str + "\n")  # Guardar todas las rutas
68
```

```
            if movimiento[-1] == 25:
                ganadoras.write(ruta_str + "\n")  # Guardar rutas ganadoras
            else:
                perdedoras.write(ruta_str + "\n")  # Guardar rutas perdedoras

# Verificar que se proporciono un argumento en la terminal
if len(sys.argv) != 2:
    print("Uso: python nombre_del_script.py <secuencia_de_colores>")
    sys.exit(1)

# Obtener la secuencia de colores del argumento
secuencia = sys.argv[1].strip().lower()

# Validar la secuencia de colores
if not all(c in ['r', 'b'] for c in secuencia):
    print("La secuencia solo debe contener 'r' para rojo y 'b' para negro.")
    sys.exit(1)

# Generar los movimientos posibles
movimientos_posibles = generar_movimientos(tablero, secuencia)

# Guardar las rutas en archivos
guardar_rutas(movimientos_posibles)

# Mostrar los movimientos posibles
if movimientos_posibles:
    print("Calculando Rutas Rey Blanco")
    #for i, movimiento in enumerate(movimientos_posibles, 1):
        #print(f"Camino {i}: {movimiento}")
else:
    print("No hay caminos posibles que respeten la secuencia de colores.")
```

Listing 3: Programa para el calculo de rutas del rey negro

```
import sys

def generar_movimientos(tablero, secuencia, posicion_inicial=5):
    movimientos = []

    # Definir el color que corresponde a 'r' y 'b'
    colores = {'r': 'rojo', 'b': 'negro'}

    # Funcion recursiva para generar las combinaciones de movimientos
    def explorar(posicion_actual, secuencia_restante, ruta):
        if not secuencia_restante:  # Si la secuencia esta vacia, guardar la ruta
            movimientos.append(ruta)
            return

        color_deseado = secuencia_restante[0]  # Color que queremos respetar en este paso
        adyacentes = tablero[posicion_actual]  # Posiciones adyacentes de la actual

        # Verifica si el color deseado esta en el diccionario
        if color_deseado not in adyacentes:
            return  # Si el color no esta, terminar la busqueda

        # Explorar cada posicion adyacente
        for adyacente in adyacentes[color_deseado]:
            # Si el color coincide, explorar mas desde esta posicion
            explorar(adyacente, secuencia_restante[1:], ruta + [adyacente])

    # Iniciar la busqueda desde la posicion inicial
    explorar(posicion_inicial, secuencia, [posicion_inicial])

    return movimientos
```

```python


# Definir posiciones adyacentes (por color) para el tablero de ajedrez de 5x5
tablero = {
    1: {'r': [2, 6], 'b': [7]},
    2: {'r': [6, 8], 'b': [1, 3, 7]},
    3: {'r': [2, 4, 8], '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]}
}

# Funcion para guardar las rutas en los archivos correspondientes
def guardar_rutas(movimientos):
    with open("Rutas/todas_las_rutasRB.txt", "w") as todas, open("Rutas/
        rutas_ganadorasRB.txt", "w") as ganadoras, open("Rutas/rutas_perdedorasRB.txt
        ", "w") as perdedoras:
        for movimiento in movimientos:
            ruta_str = " -> ".join(map(str, movimiento))  # Convertir la ruta en una
                cadena legible
            todas.write(ruta_str + "\n")  # Guardar todas las rutas

            if movimiento[-1] == 21:
                ganadoras.write(ruta_str + "\n")  # Guardar rutas ganadoras
            else:
                perdedoras.write(ruta_str + "\n")  # Guardar rutas perdedoras

# Verificar que se proporciono un argumento en la terminal
if len(sys.argv) != 2:
    print("Uso: python nombre_del_script.py <secuencia_de_colores>")
    sys.exit(1)

# Obtener la secuencia de colores del argumento
secuencia = sys.argv[1].strip().lower()

# Validar la secuencia de colores
if not all(c in ['r', 'b'] for c in secuencia):
    print("La secuencia solo debe contener 'r' para rojo y 'b' para negro.")
    sys.exit(1)

# Generar los movimientos posibles
movimientos_posibles = generar_movimientos(tablero, secuencia)

# Guardar las rutas en archivos
guardar_rutas(movimientos_posibles)

```

```
# Mostrar los movimientos posibles
if movimientos_posibles:
    print("Calculando Rutas Rey Blanco")
    #for i, movimiento in enumerate(movimientos_posibles, 1):
        #print(f"Camino {i}: {movimiento}")
else:
    print("No hay caminos posibles que respeten la secuencia de colores.")
```

Procedemos ahora con el programa para mostrar el tablero y los movimientos.

- Empezamos definiendo el ancho y largo del tablero, el número de filas y columnas, y el tamaño de cada casilla en relación con estas medidas.
- Configuramos la ventana para mostrar la cuadrícula de 5x5 en colores negro y rojo, y cargamos las imágenes correspondientes al rey blanco y negro.
- Cargamos las rutas ganadoras de ambos reyes leyéndolas desde el archivo de texto correspondiente. Elegimos una al azar para cada rey, que será la ruta que seguirá.
- En caso de no existir rutas ganadoras para algún rey, se imprimirá un aviso en la terminal.
- Elegiremos aleatoriamente qué rey empieza la partida.
- Cada rey tendrá un turno y avanzará según la siguiente posición seleccionada, calculando sus coordenadas para cada casilla.
- Si la siguiente posición del rey está ocupada por el otro, buscaremos una ruta alternativa con el criterio de que debe tener todos los pasos que ya hicimos y ser diferente a la actual, además de que el siguiente movimiento debe ser distinto al de la posición del segundo rey.
- Si el criterio anterior no se cumple, omitiremos ese turno.
- Cuando un rey llegue a la posición final habiendo seguido toda la ruta, imprimiremos qué rey ganó y su ruta ganadora.

Listing 4: Programa para mostrar el tablero y sus movimientos

```
import pygame
import sys
import random

# Inicializar Pygame
pygame.init()

# Constantes
ANCHO, ALTO = 1000, 1000
FILAS, COLUMNAS = 5, 5
TAMANO_CUADRADO = ANCHO // COLUMNAS
VELOCIDAD_MOVIMIENTO = 3  # Pixeles por cuadro para un movimiento suave

# Colores
NEGRO = (0, 0, 0)
ROJO = (255, 0, 0)

# Configurar la pantalla
VENTANA = pygame.display.set_mode((ANCHO, ALTO))
pygame.display.set_caption('Tablero de Ajedrez 5x5')

# Cargar imagenes
IMAGEN_REY1 = pygame.image.load('Imagenes/RB.png')
IMAGEN_REY2 = pygame.image.load('Imagenes/RN.png')

# Escalar imagenes para ajustarse a los cuadrados
IMAGEN_REY1 = pygame.transform.scale(IMAGEN_REY1, (TAMANO_CUADRADO, TAMANO_CUADRADO))
IMAGEN_REY2 = pygame.transform.scale(IMAGEN_REY2, (TAMANO_CUADRADO, TAMANO_CUADRADO))

```

```
def dibujar_tablero(ventana, pos_rey1, pos_rey2, rey1_x, rey1_y, rey2_x, rey2_y):
    ventana.fill(NEGRO)
    for fila in range(FILAS):
        for col in range(COLUMNAS):
            if (fila + col) % 2 == 1:
                pygame.draw.rect(ventana, ROJO, (col * TAMANO_CUADRADO, fila *
                    TAMANO_CUADRADO, TAMANO_CUADRADO, TAMANO_CUADRADO))

    # Dibujar imagenes en sus posiciones actuales
    ventana.blit(IMAGEN_REY1, (rey1_x, rey1_y))
    ventana.blit(IMAGEN_REY2, (rey2_x, rey2_y))

def cargar_rutas_de_archivo(nombre_archivo):
    with open(nombre_archivo, 'r') as archivo:
        rutas = archivo.readlines()

        # Verificar si el archivo esta vacio
        if not rutas:
            print(f"No hay rutas ganadoras en el archivo: {nombre_archivo}")
            return []  # Retorna una lista vacia si no hay rutas

    rutas = [ruta.strip().split(' -> ') for ruta in rutas]
    rutas = [[int(pos) for pos in ruta] for ruta in rutas]
    return rutas

def encontrar_ruta_alternativa(rutas, pasos_actuales, pos_rey2):
    # Busca una ruta que coincida con los pasos actuales y que no colisione con la
        posicion de rey2
    for ruta in rutas:
        if ruta[:len(pasos_actuales)] == pasos_actuales and (len(ruta) > len(
            pasos_actuales) and ruta[len(pasos_actuales)] != pos_rey2):
            return ruta
    return None

def mover_suavemente(rey_x, rey_y, objetivo_x, objetivo_y):
    direccion_x = objetivo_x - rey_x
    direccion_y = objetivo_y - rey_y
    distancia = (direccion_x ** 2 + direccion_y ** 2) ** 0.5

    if distancia == 0:
        return rey_x, rey_y

    mover_x = (direccion_x / distancia) * VELOCIDAD_MOVIMIENTO
    mover_y = (direccion_y / distancia) * VELOCIDAD_MOVIMIENTO

    if abs(mover_x) > abs(direccion_x):
        mover_x = direccion_x
    if abs(mover_y) > abs(direccion_y):
        mover_y = direccion_y

    return rey_x + mover_x, rey_y + mover_y

def main():
    reloj = pygame.time.Clock()
    ejecutar = True

    # Cargar rutas desde archivos
    rutas_rey1 = cargar_rutas_de_archivo('Rutas/rutas_ganadorasRB.txt')
    rutas_rey2 = cargar_rutas_de_archivo('Rutas/rutas_ganadorasRN.txt')

    # Verificar si hay rutas disponibles
    if not rutas_rey1 or not rutas_rey2:
        print("No se pueden iniciar las partidas debido a rutas ganadoras faltantes."
            )
        pygame.quit()
```

```
        sys.exit()

    ruta_seleccionada1 = random.choice(rutas_rey1)
    ruta_seleccionada2 = random.choice(rutas_rey2)

    paso1 = 0
    paso2 = 0

    pos_rey1 = ruta_seleccionada1[paso1]
    pos_rey2 = ruta_seleccionada2[paso2]

    fila_rey1, col_rey1 = divmod(pos_rey1 - 1, COLUMNAS)
    rey1_x, rey1_y = col_rey1 * TAMANO_CUADRADO, fila_rey1 * TAMANO_CUADRADO

    fila_rey2, col_rey2 = divmod(pos_rey2 - 1, COLUMNAS)
    rey2_x, rey2_y = col_rey2 * TAMANO_CUADRADO, fila_rey2 * TAMANO_CUADRADO

    turno_actual = random.choice([1, 2])

    pygame.display.flip() #Actualiza la pantalla

    while ejecutar:
        reloj.tick(60)
        for evento in pygame.event.get():
            if evento.type == pygame.QUIT:
                ejecutar = False

        if turno_actual == 1:
            if paso1 < len(ruta_seleccionada1) - 1:
                siguiente_pos1 = ruta_seleccionada1[paso1 + 1]

                # Si la siguiente posicion de rey1 es la actual de rey2, buscar una
                    ruta alternativa
                if siguiente_pos1 == pos_rey2:
                    ruta_alt = encontrar_ruta_alternativa(rutas_rey1,
                        ruta_seleccionada1[:paso1 + 1], pos_rey2)
                    if ruta_alt:
                        ruta_seleccionada1 = ruta_alt
                    else:
                        turno_actual = 2  # Omitir turno si no hay ruta alternativa
                        continue

                siguiente_fila1, siguiente_col1 = divmod(siguiente_pos1 - 1, COLUMNAS
                    )
                objetivo_x1, objetivo_y1 = siguiente_col1 * TAMANO_CUADRADO,
                    siguiente_fila1 * TAMANO_CUADRADO

                rey1_x, rey1_y = mover_suavemente(rey1_x, rey1_y, objetivo_x1,
                    objetivo_y1)

                if abs(rey1_x - objetivo_x1) < VELOCIDAD_MOVIMIENTO and abs(rey1_y -
                    objetivo_y1) < VELOCIDAD_MOVIMIENTO:
                    paso1 += 1
                    pos_rey1 = siguiente_pos1
                    turno_actual = 2
                    if paso1 == len(ruta_seleccionada1) - 1:
                        print('El␣Rey␣Blanco␣ha␣terminado␣su␣recorrido.')
                        print('Ruta␣ganadora:', ruta_seleccionada1)
                        ejecutar = False

        elif turno_actual == 2:
            if paso2 < len(ruta_seleccionada2) - 1:
                siguiente_pos2 = ruta_seleccionada2[paso2 + 1]

                # Si la siguiente posicion de rey2 es la actual de rey1, buscar una
```

```
                    ruta alternativa
                if siguiente_pos2 == pos_rey1:
                    ruta_alt = encontrar_ruta_alternativa(rutas_rey2,
                        ruta_seleccionada2[:paso2 + 1], pos_rey1)
                    if ruta_alt:
                        ruta_seleccionada2 = ruta_alt
                    else:
                        turno_actual = 1  # Omitir turno si no hay ruta alternativa
                        continue

                siguiente_fila2, siguiente_col2 = divmod(siguiente_pos2 - 1, COLUMNAS
                    )
                objetivo_x2, objetivo_y2 = siguiente_col2 * TAMANO_CUADRADO,
                    siguiente_fila2 * TAMANO_CUADRADO

                rey2_x, rey2_y = mover_suavemente(rey2_x, rey2_y, objetivo_x2,
                    objetivo_y2)

                if abs(rey2_x - objetivo_x2) < VELOCIDAD_MOVIMIENTO and abs(rey2_y -
                    objetivo_y2) < VELOCIDAD_MOVIMIENTO:
                    paso2 += 1
                    pos_rey2 = siguiente_pos2
                    turno_actual = 1
                    if paso2 == len(ruta_seleccionada2) - 1:
                        print('El Rey Negro ha terminado su recorrido.')
                        print('Ruta ganadora:', ruta_seleccionada2)
                        ejecutar = False

        dibujar_tablero(VENTANA, pos_rey1, pos_rey2, rey1_x, rey1_y, rey2_x, rey2_y)
        pygame.display.flip()

    pygame.quit()

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

Por último, tenemos los dos programas para graficar las redes.

- Leemos el archivo `txt` de todas las rutas, separamos los números individualmente y los convertimos en una matriz.
- Eliminamos los elementos repetidos de cada columna de la matriz.
- Eliminamos las filas vacías que pudieron haber quedado.
- Graficamos la matriz resultante como si fueran celdas, y en cada celda irá el número correspondiente.
- Dibujamos las líneas según las rutas de nuestro `txt`, indicando en qué posición inicia y termina cada línea.
- La diferencia entre un programa y otro es el título de la gráfica y el archivo que lee.

Listing 5: Programa para mostrar la red de movimientos del rey blanco

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patheffects import withStroke
import random

def leer_archivo_a_matriz(nombre_archivo):
    # Inicializa una lista para almacenar las filas
    matriz = []

    # Abre el archivo en modo lectura
    with open(nombre_archivo, 'r') as archivo:
        # Lee cada linea del archivo
        for linea in archivo:
```

```
            # Elimina espacios y separa por '->', luego convierte a enteros
            fila = [int(x.strip()) for x in linea.split('->')]
            # Agrega la fila a la matriz
            matriz.append(fila)

    return matriz

def eliminar_repetidos(matriz):
    # Convertir la matriz a un array de numpy con dtype float
    matriz_np = np.zeros_like(matriz, dtype=float)  # Inicializa con ceros
    matriz_np[:] = np.nan  # Llena la matriz con NaN

    # Obtener el numero de columnas
    num_columnas = matriz_np.shape[1]

    # Iterar sobre cada columna
    for col in range(num_columnas):
        elementos_vistos = set()  # Conjunto para almacenar elementos ya vistos
        for fila in range(matriz_np.shape[0]):  # Cambiado para evitar len
            valor = matriz[fila][col]
            if valor in elementos_vistos:
                matriz_np[fila, col] = np.nan  # Reemplaza el valor por NaN
            else:
                elementos_vistos.add(valor)  # Agrega el nuevo elemento al conjunto
                matriz_np[fila, col] = valor  # Guarda el valor en la nueva matriz

    return matriz_np

def filtrar_filas_vacias(matriz):
    """Elimina filas que solo contienen NaN."""
    return matriz[~np.all(np.isnan(matriz), axis=1)]

def generar_color_aleatorio():
    """Genera un color aleatorio en formato RGB."""
    return (random.random(), random.random(), random.random())

def graficar_matriz(matriz, rutas):
    # Obtiene las dimensiones de la matriz
    filas, columnas = matriz.shape

    # Configura el tamano de la figura en funcion de las dimensiones de la matriz
    plt.figure(figsize=(2 * columnas, 2 * filas))  # Ajusta el tamano de la figura

    # Itera sobre cada celda y coloca el texto correspondiente
    for i in range(filas):
        for j in range(columnas):
            valor = matriz[i, j]
            if not np.isnan(valor):  # Solo grafica si el valor no es NaN
                texto = plt.text(j, i, str(int(valor)), ha='center', va='center',
                    fontsize=8, color='black',path_effects=[withStroke(linewidth=7,
                    foreground='white')])

    # Configura el grafico como una cuadricula
    plt.xlim(-0.5, columnas - 0.5)
    plt.ylim(filas - 0.5, -0.5)
    plt.xticks(np.arange(columnas), labels=np.arange(1, columnas + 1), rotation=45)
        # Rotar etiquetas
    plt.yticks(np.arange(filas), labels=np.arange(1, filas + 1))
    plt.grid(True, which='both', color='black', linestyle='-', linewidth=1, alpha
        =0.5)

# Dibuja las lineas segun las rutas
    for ruta in rutas:
        color = generar_color_aleatorio()  # Genera un color aleatorio solo una vez
            para cada ruta
```

```
        for j in range(np.shape(ruta)[0] - 1):  # Cambiado para evitar len
            y_start = np.where(matriz[:, j] == ruta[j])[0]  # Fila del numero de inicio
            y_end = np.where(matriz[:, j + 1] == ruta[j + 1])[0]  # Fila del numero de fin
            if y_start.size > 0 and y_end.size > 0:  # Si ambos numeros estan en la matriz
                plt.plot([j, j + 1], [y_start[0], y_end[0]], color=color, linestyle='-', linewidth=2)  # Dibuja la linea


    # Agrega un titulo
    plt.title('Red Rey Blanco')
    plt.xlabel('')
    plt.ylabel('')

    # Muestra la grafica
    plt.gca().set_aspect('auto', adjustable='box')
    plt.grid(False)
    plt.show()

# Nombre del archivo de texto
nombre_archivo = 'Rutas/todas_las_rutasRB.txt'

# Llama a la funcion para leer el archivo y obtener la matriz
matriz_resultante = leer_archivo_a_matriz(nombre_archivo)

# Elimina los repetidos en la matriz
matriz_final = eliminar_repetidos(matriz_resultante)

# Filtra las filas vacias
matriz_final = filtrar_filas_vacias(matriz_final)

# Grafica la matriz final
graficar_matriz(matriz_final, matriz_resultante)
```

Listing 6: Programa para mostrar la red de movimientos del rey negro

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patheffects import withStroke
import random

def leer_archivo_a_matriz(nombre_archivo):
    # Inicializa una lista para almacenar las filas
    matriz = []

    # Abre el archivo en modo lectura
    with open(nombre_archivo, 'r') as archivo:
        # Lee cada linea del archivo
        for linea in archivo:
            # Elimina espacios y separa por '->', luego convierte a enteros
            fila = [int(x.strip()) for x in linea.split('->')]
            # Agrega la fila a la matriz
            matriz.append(fila)

    return matriz

def eliminar_repetidos(matriz):
    # Convertir la matriz a un array de numpy con dtype float
    matriz_np = np.zeros_like(matriz, dtype=float)  # Inicializa con ceros
    matriz_np[:] = np.nan  # Llena la matriz con NaN

    # Obtener el numero de columnas
    num_columnas = matriz_np.shape[1]
```

```
28
29      # Iterar sobre cada columna
30      for col in range(num_columnas):
31          elementos_vistos = set()  # Conjunto para almacenar elementos ya vistos
32          for fila in range(matriz_np.shape[0]):  # Cambiado para evitar len
33              valor = matriz[fila][col]
34              if valor in elementos_vistos:
35                  matriz_np[fila, col] = np.nan  # Reemplaza el valor por NaN
36              else:
37                  elementos_vistos.add(valor)  # Agrega el nuevo elemento al conjunto
38                  matriz_np[fila, col] = valor  # Guarda el valor en la nueva matriz
39
40      return matriz_np
41
42  def filtrar_filas_vacias(matriz):
43      """Elimina filas que solo contienen NaN."""
44      return matriz[~np.all(np.isnan(matriz), axis=1)]
45
46  def generar_color_aleatorio():
47      """Genera un color aleatorio en formato RGB."""
48      return (random.random(), random.random(), random.random())
49
50  def graficar_matriz(matriz, rutas):
51      # Obtiene las dimensiones de la matriz
52      filas, columnas = matriz.shape
53
54      # Configura el tamano de la figura en funcion de las dimensiones de la matriz
55      plt.figure(figsize=(2 * columnas, 2 * filas))  # Ajusta el tamano de la figura
56
57      # Itera sobre cada celda y coloca el texto correspondiente
58      for i in range(filas):
59          for j in range(columnas):
60              valor = matriz[i, j]
61              if not np.isnan(valor):  # Solo grafica si el valor no es NaN
62                  texto = plt.text(j, i, str(int(valor)), ha='center', va='center',
                        fontsize=8, color='black',path_effects=[withStroke(linewidth=7,
                        foreground='white')])
63
64      # Configura el grafico como una cuadricula
65      plt.xlim(-0.5, columnas - 0.5)
66      plt.ylim(filas - 0.5, -0.5)
67      plt.xticks(np.arange(columnas), labels=np.arange(1, columnas + 1), rotation=45)
            # Rotar etiquetas
68      plt.yticks(np.arange(filas), labels=np.arange(1, filas + 1))
69      plt.grid(True, which='both', color='black', linestyle='-', linewidth=1, alpha
            =0.5)
70
71  # Dibuja las lineas segun las rutas
72      for ruta in rutas:
73          color = generar_color_aleatorio()  # Genera un color aleatorio solo una vez
                para cada ruta
74          for j in range(np.shape(ruta)[0] - 1):  # Cambiado para evitar len
75              y_start = np.where(matriz[:, j] == ruta[j])[0]  # Fila del numero de
                    inicio
76              y_end = np.where(matriz[:, j + 1] == ruta[j + 1])[0]  # Fila del numero
                    de fin
77              if y_start.size > 0 and y_end.size > 0:  # Si ambos numeros estan en la
                    matriz
78                  plt.plot([j, j + 1], [y_start[0], y_end[0]], color=color, linestyle='
                        -', linewidth=2)  # Dibuja la linea
79
80
81      # Agrega un titulo
82      plt.title('Red␣Rey␣Negro')
83      plt.xlabel('')
```

```
    plt.ylabel('')

    # Muestra la grafica
    plt.gca().set_aspect('auto', adjustable='box')
    plt.grid(False)
    plt.show()

# Nombre del archivo de texto
nombre_archivo = 'Rutas/todas_las_rutasRN.txt'

# Llama a la funcion para leer el archivo y obtener la matriz
matriz_resultante = leer_archivo_a_matriz(nombre_archivo)

# Elimina los repetidos en la matriz
matriz_final = eliminar_repetidos(matriz_resultante)

# Filtra las filas vacias
matriz_final = filtrar_filas_vacias(matriz_final)

# Grafica la matriz final
graficar_matriz(matriz_final, matriz_resultante)
```

Resultados

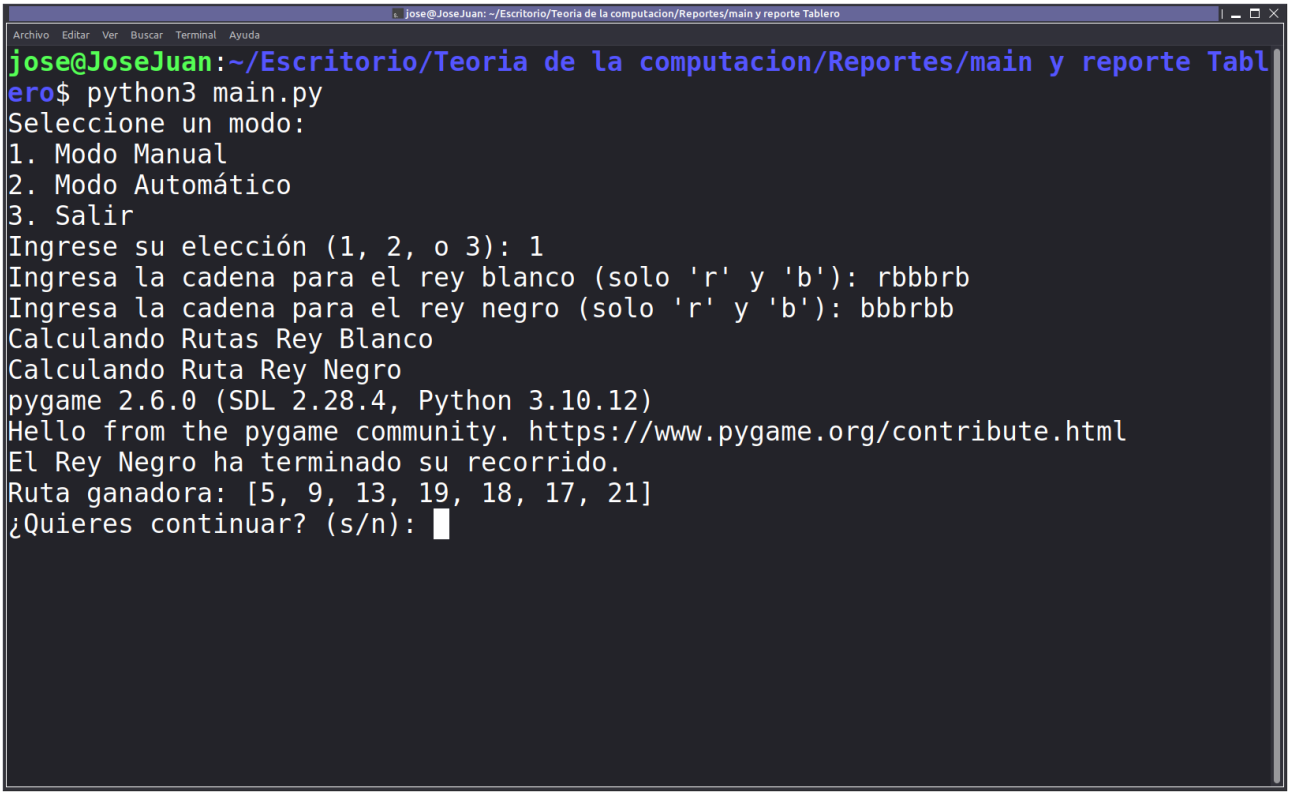

Figure 1: Ejecución Manual

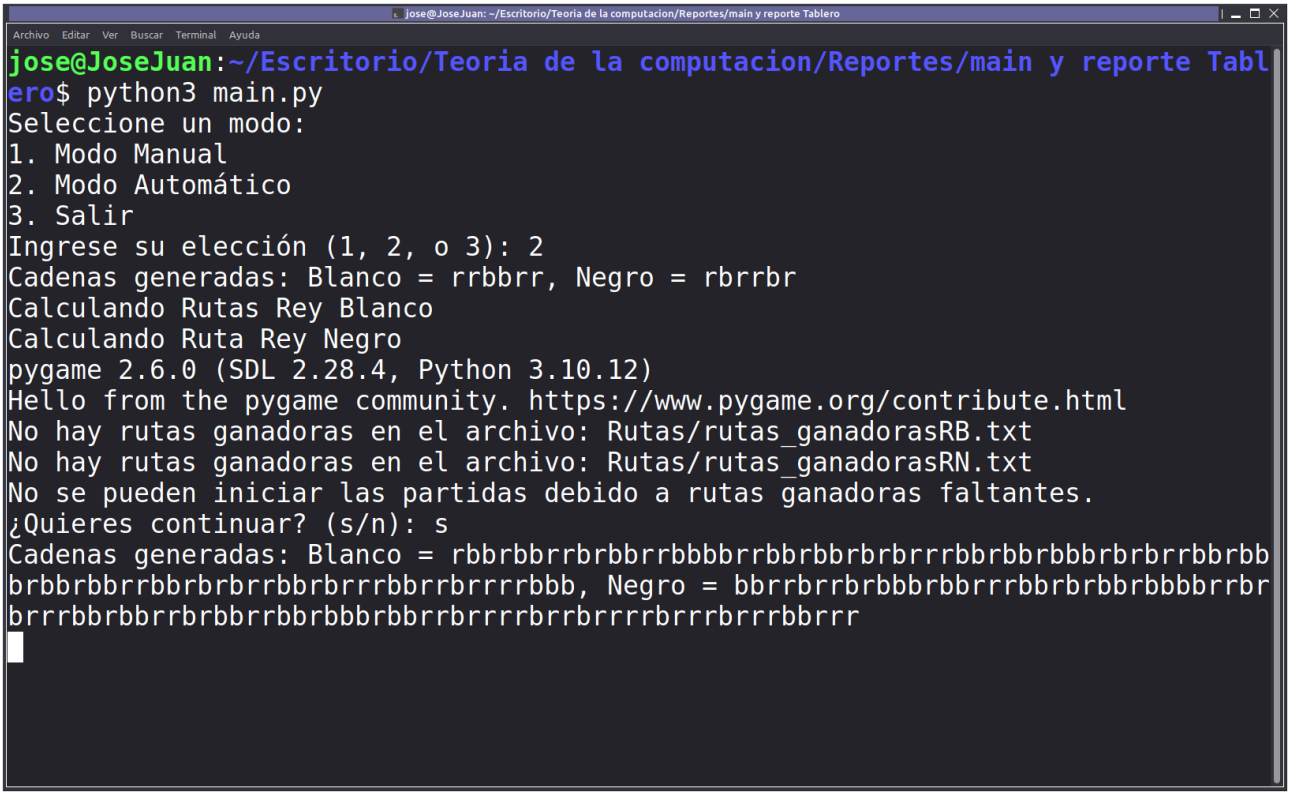

Figure 2: Ejecución Automatica

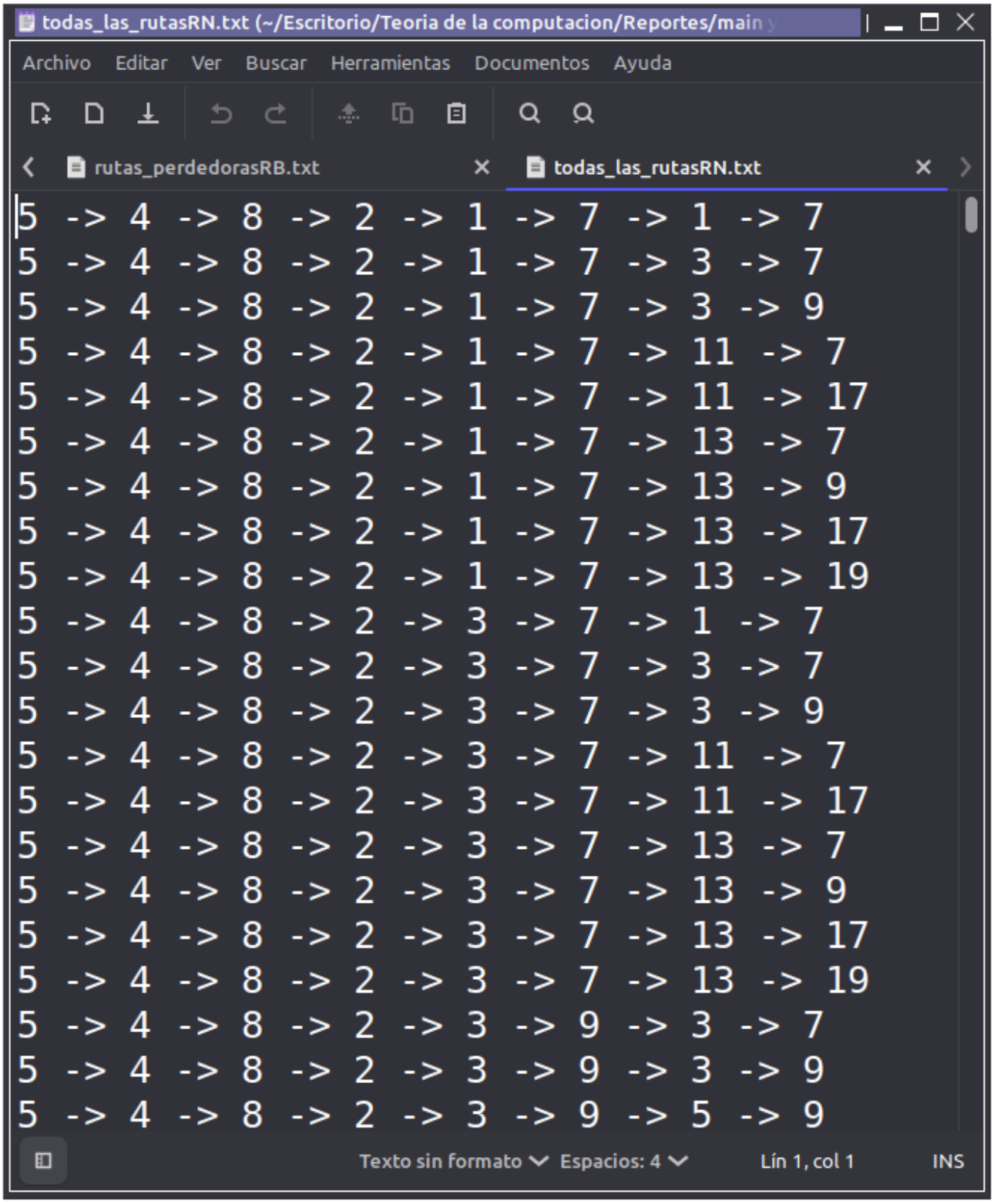

Figure 3: Txt todas las rutas

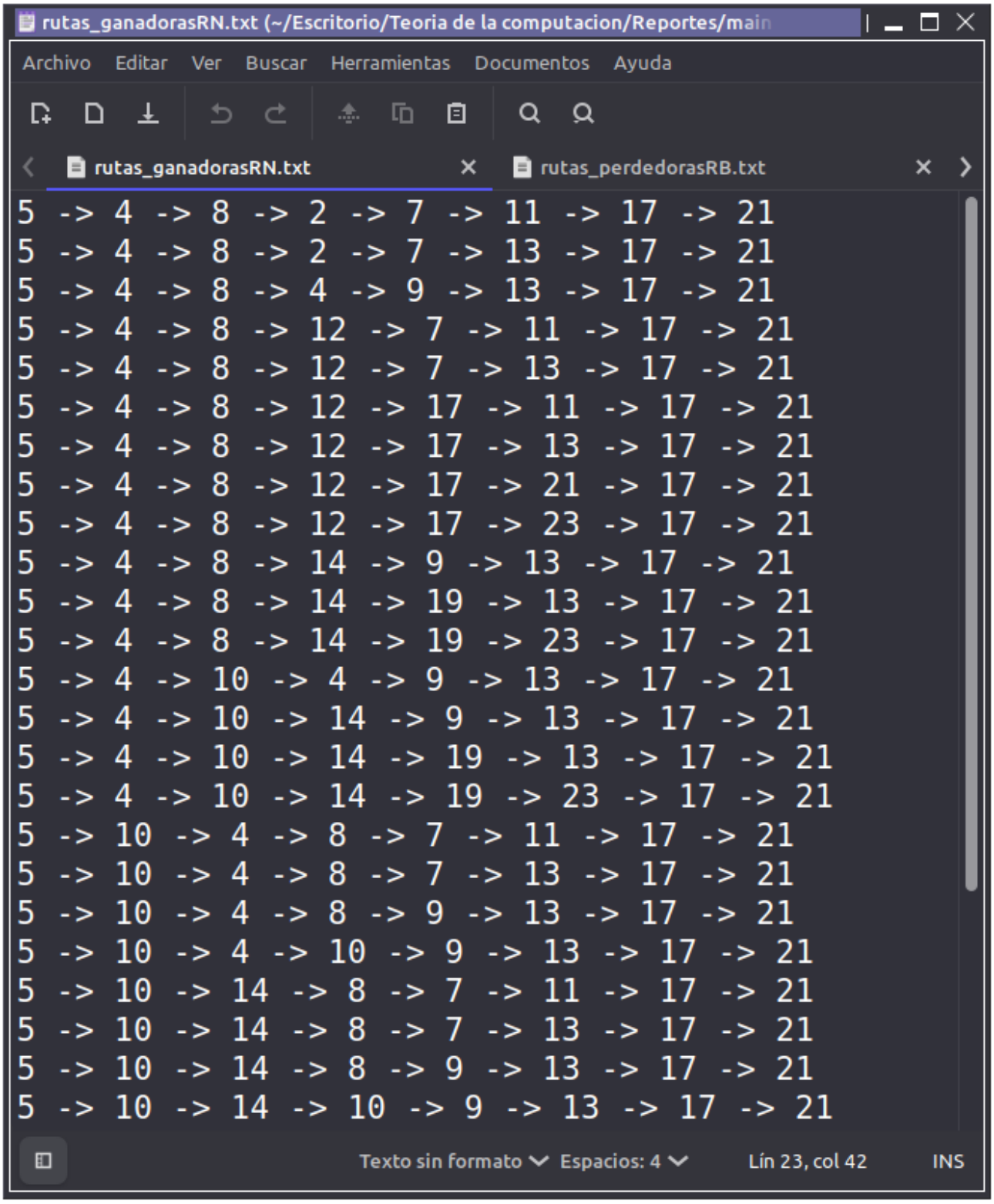

Figure 4: Txt rutas ganadoras

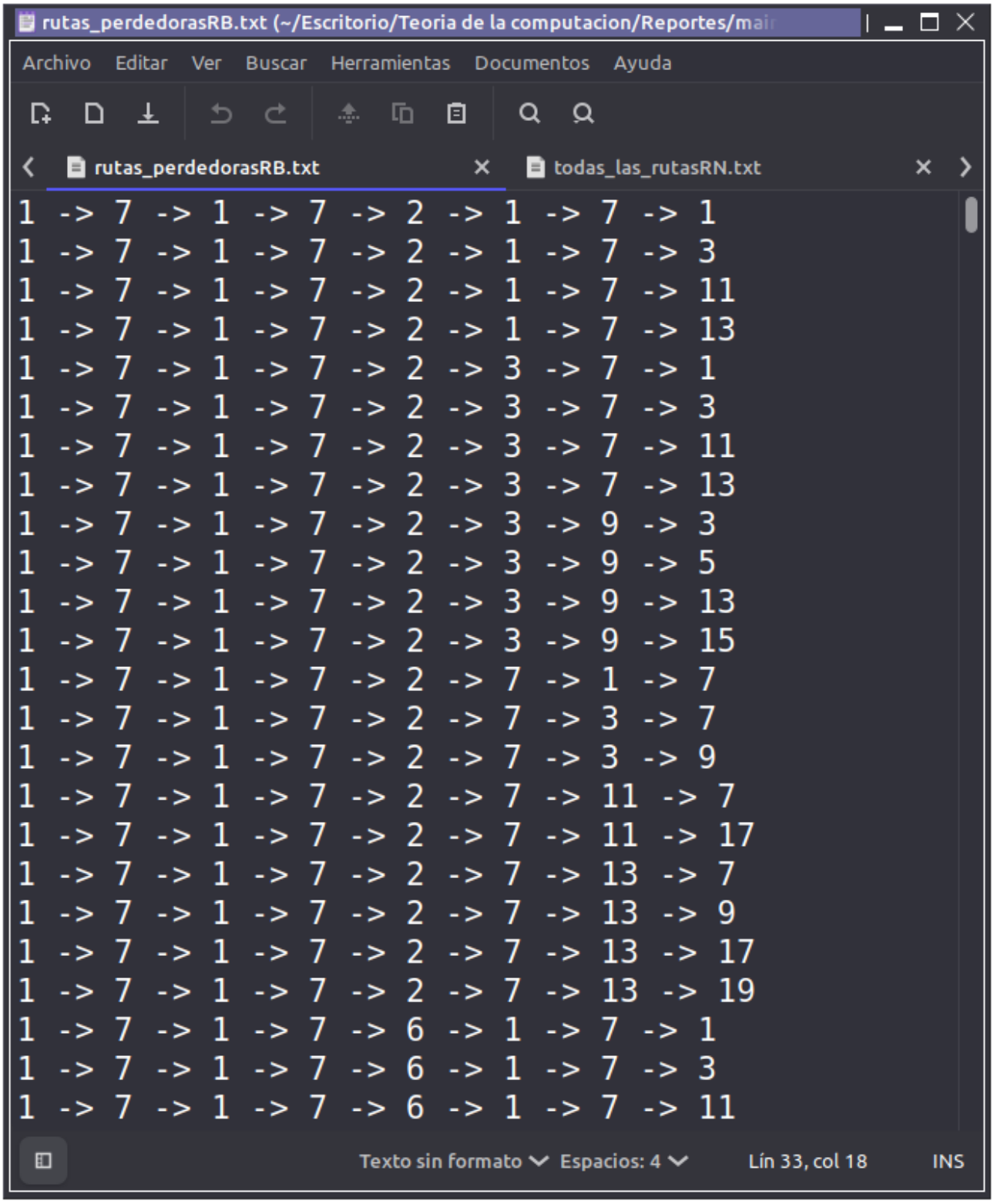

Figure 5: Txt rutas perdedoras

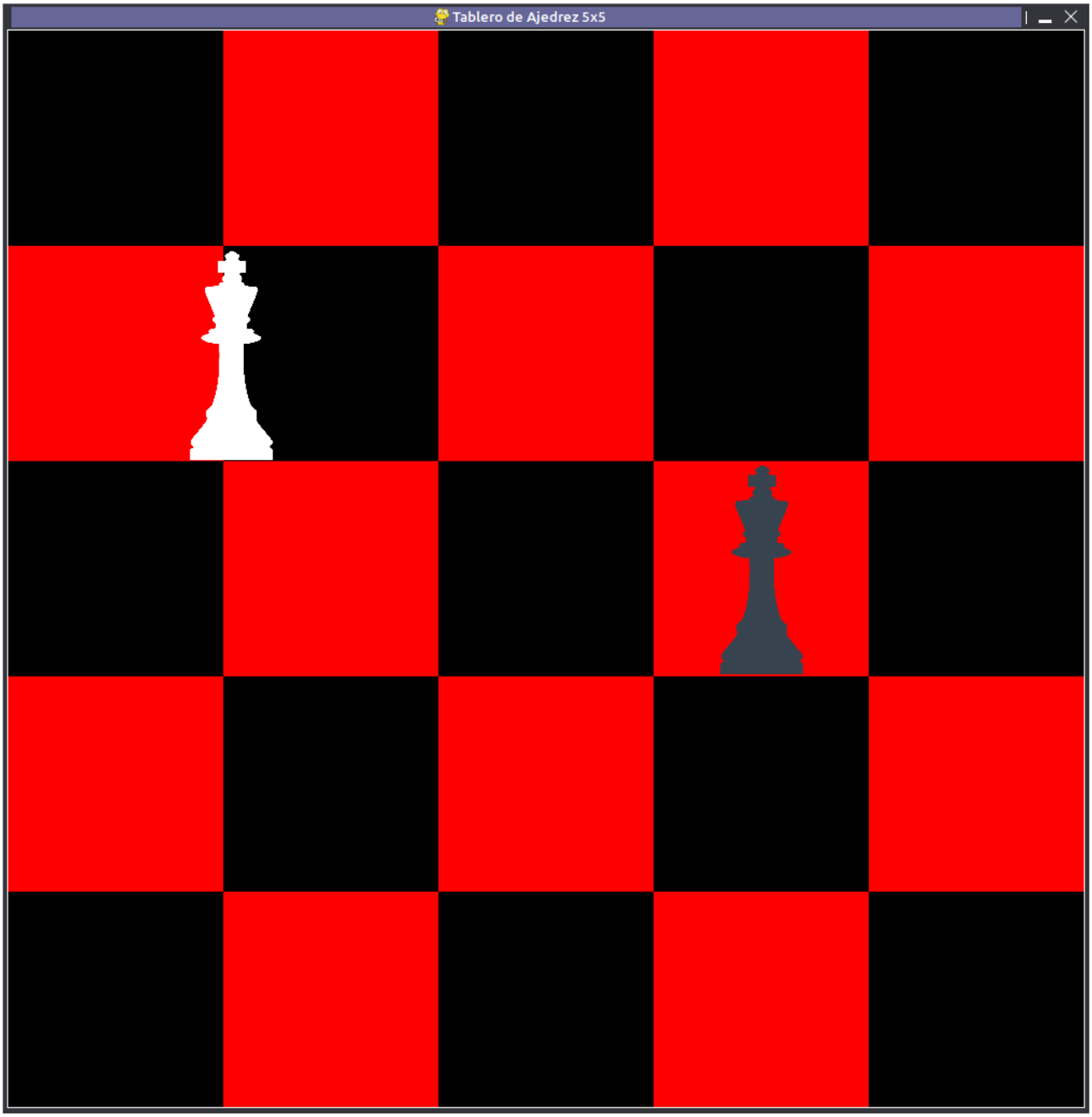

Figure 6: Ejecucion del tablero

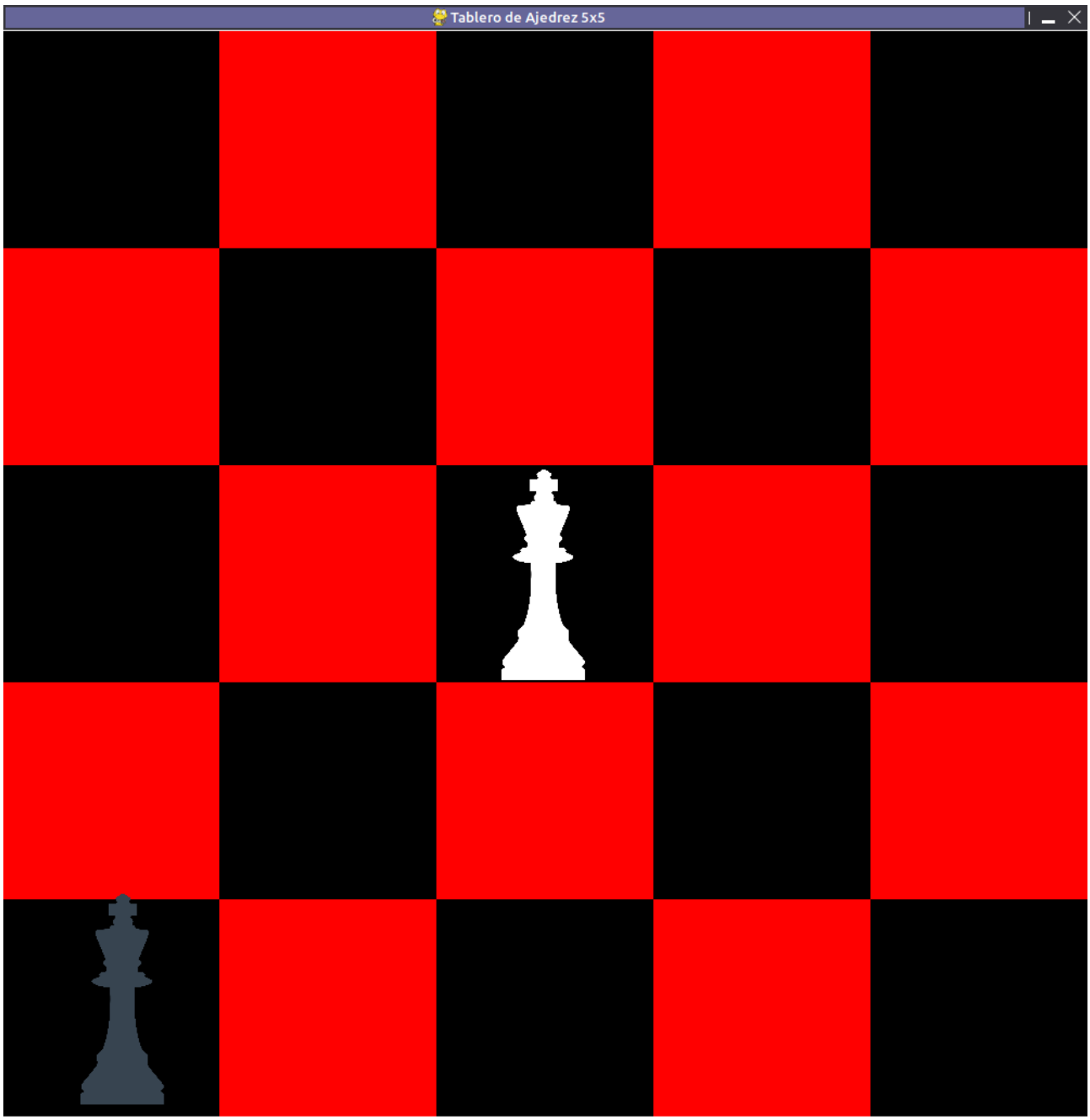

Figure 7: Ejecucion del tablero

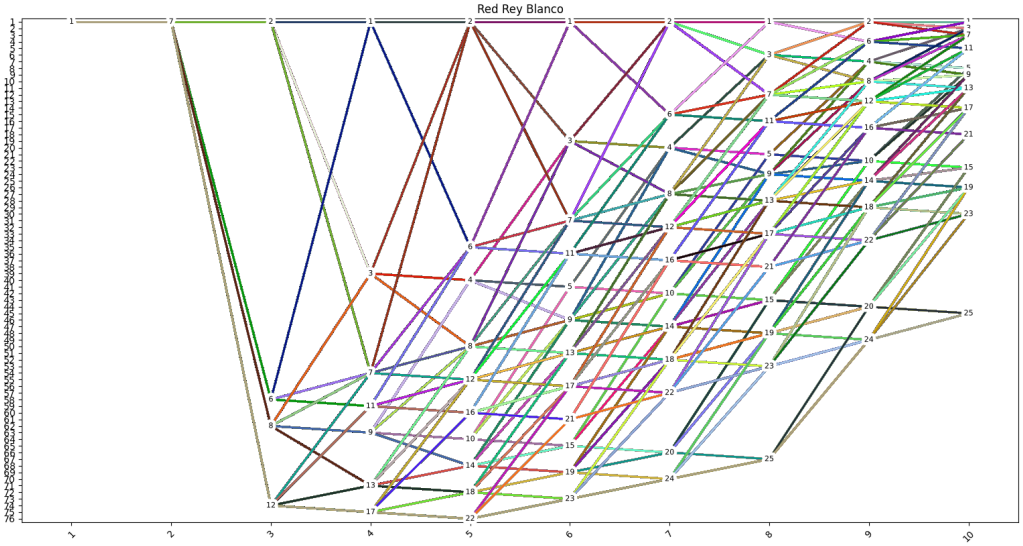

Figure 8: Red rey blanco para cadena de longitud 10

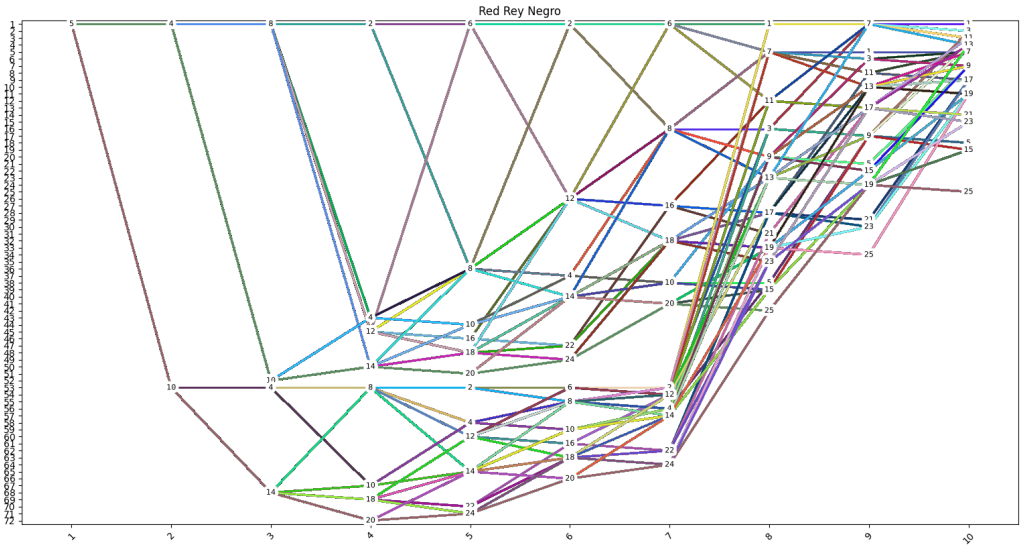

Figure 9: Red rey negro para cadena de longitud 10

Conclusiones

Elaborar este autómata me permitió profundizar sobre la teoría de autómatas y grafos, entendí mejor el funcionamiento de las transiciones y los comportamientos determinísticos. Aprendí a adaptar un NFA a diversas configuraciones y a representarlo gráficamente, llevando la parte teórica a algo más visual y práctico, además que me ha permitido visualizar esto en otras aplicaciones para resolver problemas.