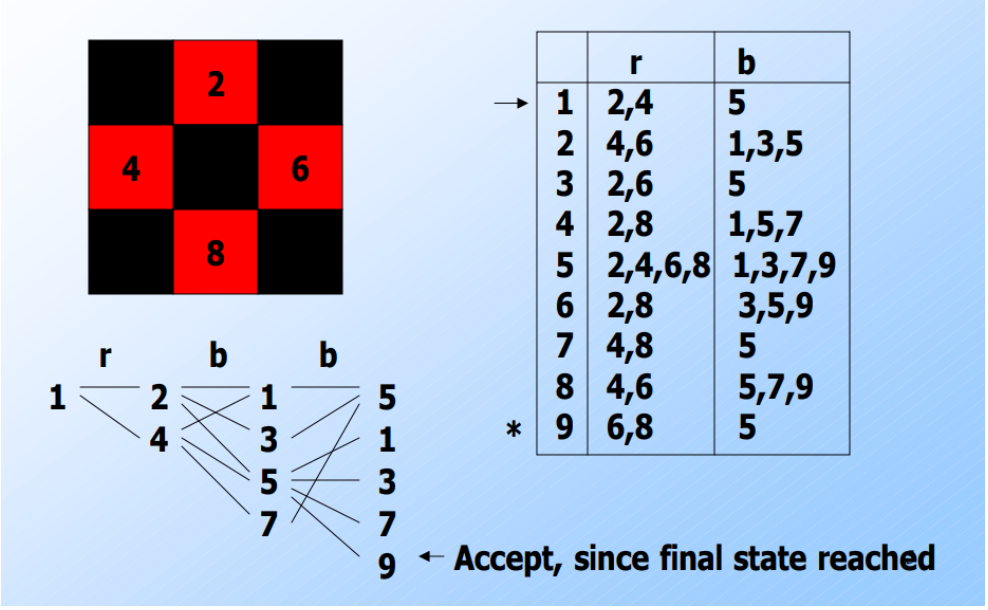
Tablero

Sebastián Acevedo Juárez
Teoría de la computación

June 2022

1 Introducción

El problema se nos plantea de la siguiente manera: En un tablero de 4x4, colocar dos piezas que tienen el comportamiento de un rey. El primero se colocará en la esquina superior izquierda, teniendo como objetivo llegar a la esquina inferior derecha. El segundo, en la esquina superior derecha para llegar a la esquina inferior izquierda. Cada casilla del tablero representa un estado. Para determinar la ruta que debe seguir cada pieza, el usuario o el programa en su defecto, debe de ingresar una cadena de r's y b's. Esta representa el color rojo y negro respectivamente. Cada vez que en la cadena encontremos una r, significa que la ficha se tiene que mover en alguna de las casillas rojas adyacentes a su posición actual. Aunado a esto, las fichas nos deben de cruzar caminos en ninguna de las casillas del tablero.

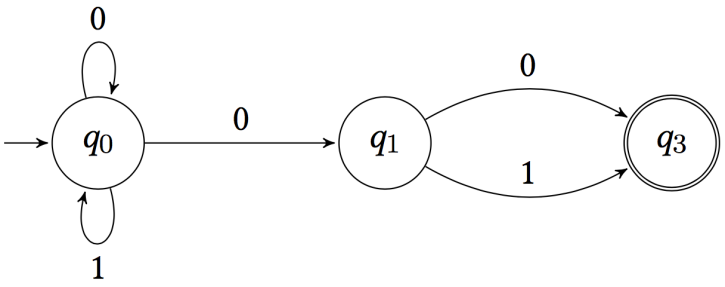


		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

2 Marco Teórico

Un **autómata finito no determinista**(NFA o AFN por sus siglas en inglés y español respectivamente), es el autómata finito que tiene transiciones vacías o que por cada símbolo desde un estado de origen se llega a más de un estado destino, es decir, es aquel que, a diferencia de los autómatas finitos deterministas, posee al menos un estado , tal que para un símbolo del alfabeto, existe más de una transición posible. En palabras más simples, en un NFA tenemos varios caminos posibles dada una condición. Un Autómata Finito No Determinista consta de:

1. Un conjunto finito de estados, a menudo designado como Q.
2. Un conjunto finito de símbolos de entrada, a menudo designado como Σ (sigma).
3. Una función de transición que toma como argumentos un estado y un símbolo de entrada y devuelve un estado. La función de transición se designa habitualmente como δ o Δ (delta).
4. Un estado inicial, uno de los estados de Q.
5. Un conjunto de estados finales o de aceptación F. El conjunto F es un subconjunto de Q.



Representación gráfica de un NFA

Para resolver nuestro ejercicio, tenderemos que convertir un NFA a DFA. Los AFN son útiles como herramienta teórica: facilitan el diseño y suelen ser más "económicos" (menor numero de estados, menor número de transiciones) que los AFD. Pero, por desgracia, son herramientas teóricas: el mundo real es determinista, por lo tanto, para poder "verlos en funcionamiento" o bien se recurre a la simulación o bien se deben transformar previamente a AFD.

1. Se construye una tabla donde cada columna esta etiquetada con un símbolo del alfabeto de entrada y cada fila se etiqueta con un conjunto de estados.

2. La primera fila se etiqueta con $q0$, estado inicial, y en cada entrada de la tabla $[q0, si]$ se almacena $f(q0, si) = p1, ..., pn = P$.

3. Se etiqueta cada fila con cada uno de los conjuntos P que no tengan asociada una fila en la tabla (es decir, con cada uno de los conjuntos P que aparezcan por primera vez en la tabla) y se completa cada una de estas filas con el correspondiente $f(P, si)$.

4. Se realiza el paso (3) hasta que no haya en la tabla conjuntos P sin filas asociadas.

5. Se asocia a cada conjunto P que aparezca en la tabla un estado en el nuevo AFD y aquellos que tengan entre sus componentes algún estado final del AFN se considerarán estados finales en el AFD.

3 Desarrollo

Para el desarrollo del programa, únicamente están disponibles las implementaciones de 1 y 2 fichas con una cadena generada automáticamente. Al terminar la ejecución, el programa genera un archivo txt con el camino a considerar toamndo en cuenta la cadena generada. Mediante los resultados arrojados pdemos comprobar que para el caso donde se utilizan 2 fichas, ninguna choca con la otra en alguno de sus estados.

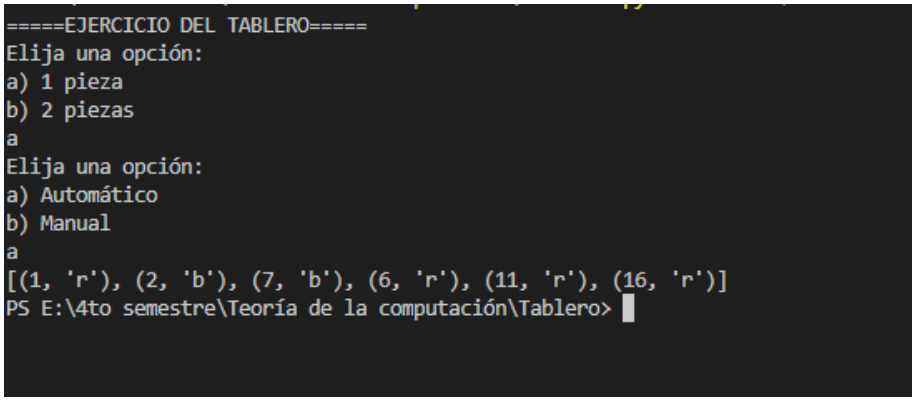
```
import random
import math

def main():
    #Crea la matriz del tablero con arreglos
    matriz = crearMatriz()
    while(1):
        print("=====EJERCICIO DEL TABLERO=====")
        print("Elija una opci n:")
        print("a) 1 pieza")
        print("b) 2 piezas")
        piezas = input()
        print("Elija una opci n:")
        print("a) Autom tico")
        print("b) Manual")
        modo = input()

        if(piezas == "a"):
            if(modo == "a"): #1 pieza automatico
                unaATM(matriz)
                break
            elif(modo == "b"): # 1 pieza manual
                unaMAN(matriz)
                break
            else: print("Opci n de modo no disponible ")
        elif(piezas == "b"):
            if(modo == "a"): #2 piezas automatico
                dosATM(matriz)
                break
```

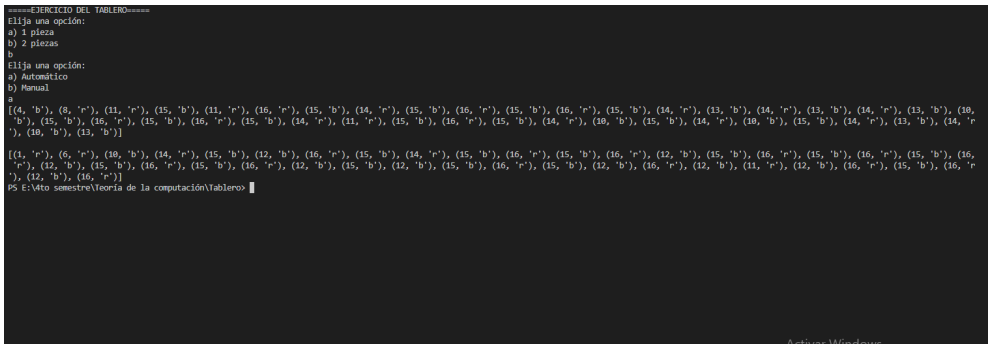
```
            elif(modo == "b"): # 2 piezas manual
                break
            else: print("Opci n de modo no disponible" )
        else: print("Opci n de pieza no disponible" )

def crearMatriz():
    colores=['r','b','r','b']

    #Crea matriz de 4x4
    # 4 filas de 4 elementos
    matriz = [[(x+1+y*4) for x in range(4)] for y in range(4)]

    for i in range(len(matriz)):
        '''
        Asigna el color de la casilla. Usa el m todo zip para combinar las filas con la lista coloroes.
        El m todo list hace que retorne una lista, para que la matriz no se vea alterada.
        '''
        matriz[i] = list(zip(matriz[i],colores))

        #Cada vez que hay un salto de fila, el orden de los colores se invierte.
        colores.reverse()
    return(matriz)

def unaATM(matriz):
    path = [matriz[0][0]]
    ficha = 1
    while(ficha != 16 ):

        i = int(math.ceil(ficha/4)) - 1
        j = ficha - (i*4) - 1
        moves = crearMoves(i,j,matriz,ficha)

        peso = [(d+1)*(2*d+3) for d in range(len(moves))]
        siguiente = random.choices(moves, weights=peso, k=1)[0]
        ficha = siguiente[0]
        moves.clear()
        path.append(siguiente)

    print(path)

def unaMAN(matriz):
    str=[]
    print("Ingrese la cadena por colores")
    str[:0]= input()
    path = []
    ficha = 1

    while(ficha != 16 ):
        i = int(math.ceil(ficha/4)) - 1
        j = ficha - (i*4) - 1
        moves = crearMoves(i,j,matriz,ficha)

        peso = [(d+1)*(2*d*d) for d in range(len(moves))]
        siguiente = random.choices(moves, weights=peso, k=1)
```

```
84          ficha = siguiente[0][0]
85          moves.clear()
86          path.append(siguiente)
87      print(path)
88
89  def dosATM(matriz):
90      path1 = [matriz[0][0]]
91      path2 = [matriz[0][3]]
92      lim1 = 16
93      lim2 = 13
94      ficha1 = 1
95      ficha2 = 4
96
97      if(random.randint(1, 2) == 2):
98          path1,path2 = path2,path1
99          ficha1,ficha2 = ficha2, ficha1
100         lim1,lim2 = lim2,lim1
101
102     while(ficha1 != lim1 or ficha2 != lim2):
103
104         i = int(math.ceil(ficha1/4)) - 1
105         j = ficha1 - (i*4) - 1
106         moves1 = crearMoves(i,j,matriz,ficha1)
107         #print(moves1)
108
109         i = int(math.ceil(ficha2/4))-1
110         j = ficha2 - (i*4) - 1
111         moves2 = crearMoves(i,j,matriz,ficha2)
112         #print(moves2)
113
114         peso = [(d+1)*(2*d) for d in range(len(moves1))]
115         #peso = [(d+1)*40 for d in range(len(moves1))]
116         #print(peso)
117
118         siguiente = random.choices(moves1, weights=peso, k=1)[0]
119         ficha1 = siguiente[0]
120         path1.append(siguiente)
121         #print(siguiente)
122         moves1.clear()
123
124         peso = [(d+1)*(2*d) for d in range(len(moves2))]
125         #peso = [(d+1)*40 for d in range(len(moves2))]
126
127         while(siguiente[0] == ficha1):
128             siguiente = random.choices(moves2, weights=peso, k=1)
    [0]
129             ficha2 = siguiente[0]
130             #print(ficha2)
131
132         path2.append(siguiente)
133
134         moves2.clear()
135     print(path1,"\n")
136     print(path2)
137
138 def crearMoves(i,j,matriz,ficha):
139     moves = []
```

```
    for k in range(i-1,i+2):
        if(-1 < k < 4):
            for m in range(j-1,j+2):
                if(-1 < m < 4):
                    if (matriz[k][m][0] != ficha):
                        moves.append(matriz[k][m])
    return moves


main()
```

Algoritmo 1: Implementación de DFA





4 Conclusión

Este sin duda, fue el programa más complicado de todos. No sólo requiere tener un conocimiento claro de lo que es un NFA, si no también pensar detalladamente como se moverían las piezas dentro del tablero de ajedrez. Es indispensable tener un buen entendimiento abstracto para plantear una solución y del mismo modo implementarla. Una cadena puede tener varios estados finales, lo que lo hace aún más complicado para poder arrojar una respuesta definitiva. En general, me parece un muy buen programa para implementar los conocimients de la clase, sólo que requiere mucho tiempo de planeación.

References

[1] niversidad de Stanford. (2009). CS154: Introduction to Automata and Complexity Theory. CS154: Introduction to Automata and Complexity Theory. Recuperado 5 de junio de 2022, de http://infolab.stanford.edu/