

3.- Protocolo (Automata de Paridad Binaria)

Felipe Sánchez Martínez
Escuela Superior de Cómputo, IPN

Septiembre de 2023

Índice general

1. Planteamiento del problema.	1
2. Marco Teórico.	3
2.0.1. El Protocolo	3
3. Desarrollo del problema.	5
3.0.1. Approach.	5
3.0.2. Complejidad	7
3.0.3. Código fuente.	8
4. Conclusión.	19
5. Referencias.	21

Planteamiento del problema.

Una de las características principales de los autómatas finitos es que su forma de control es "determinista". Esto significa que el autómata debe estar en un estado a la vez y no, como los "no deterministas", que pueden encontrarse en más de un estado al mismo tiempo. Cabe resaltar que el hecho de tener un autómata no determinista no significa que pueda resolver problemas que un determinista no pueda; sin embargo, puede existir una eficiencia al resolver un problema con un autómata no determinista. Una representación clásica de un autómata determinista son los protocolos.

Un protocolo se refiere a un conjunto de reglas y convenciones que rigen la comunicación, la transmisión de datos y la interacción entre dispositivos, sistemas o aplicaciones en una red de computadoras. Los protocolos son esenciales para garantizar que los dispositivos puedan comunicarse entre sí de manera efectiva y comprender cómo deben intercambiar información. Además, dentro de un protocolo, cada estado representa una situación en la que uno de los participantes puede estar.

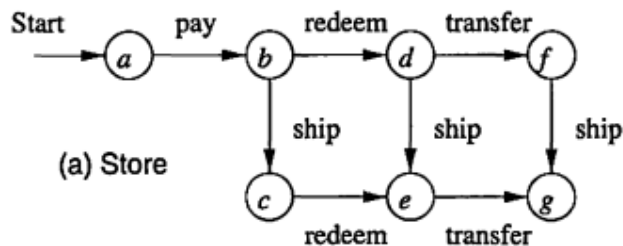


Figura 1.1: Representacion de un Automata Finito en una tienda.

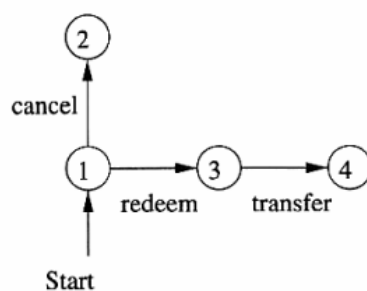
Dentro de la práctica se planea generar dos autómatas: uno finito determinista y uno infinito determinista. El autómata finito determinista es un autómata de paridad par en el número de cadenas de bits, mientras que el autómata infinito será un autómata de protocolo. Este último generará las cadenas de bits y llamará al segundo autómata que las procesará para determinar si son cadenas válidas o inválidas según la paridad par. El primer autómata funcionará siempre y cuando esté activo.

7 bits de datos	byte con bit de paridad	
	par	impar
0000000	00000000	00000001
1010001	10100011	10100010
1101001	11010010	11010011
1111111	11111111	11111110

Figura 1.2: Paridad de bits.

2.0.1. El Protocolo

Los protocolos se pueden representar como autómatas finitos. Cada estado representa una situación en la que uno de los participantes podría encontrarse. Es decir, el estado recuerda que ciertos eventos importantes han ocurrido y que otros aún no han sucedido. Las transiciones entre estados ocurren cuando se producen uno de los cinco eventos descritos anteriormente. Pensaremos en estos eventos como .^{externos}a los autómatas que representan a los tres participantes, aunque cada participante es responsable de iniciar uno o más de los eventos. Resulta que lo importante del problema es qué secuencias de eventos pueden ocurrir, no quién tiene permitido iniciarlos.



(c) Bank

Figura 2.1: Protocolo del banco.

Examinemos primero el autómata para el banco. El estado inicial es el estado 1; representa la situación en la que el banco ha emitido el archivo de dinero en cuestión pero aún no ha sido solicitado para redimirlo o cancelarlo. Si un cliente o una tienda envía una solicitud de cancelación al banco, el banco devuelve el dinero a la cuenta del cliente y entra en el estado 2. Este último estado representa la situación en la que el dinero ha sido cancelado. El banco, siendo responsable, no abandonará el estado 2 una vez que se haya ingresado, ya que no debe permitir que el mismo dinero

sea cancelado nuevamente o gastado por el cliente.

Alternativamente, cuando el banco está en el estado 1, puede recibir una solicitud de redención por parte de la tienda. Si es así, pasa al estado 3 y poco después envía a la tienda un mensaje de transferencia, con un nuevo archivo de dinero que ahora pertenece a la tienda. Después de enviar el mensaje de transferencia, el banco pasa al estado 4. En ese estado, no aceptará solicitudes de cancelación o redención ni realizará ninguna otra acción con respecto a este archivo de dinero en particular.

Desarrollo del problema.

3.0.1. Approach.

Se decidió implementar el algoritmo utilizando un ciclo 'for', para crear las 'k' cadenas de 'b' bits, y añadiendo un automata de paridad, que lea todas las cadenas y termine clasificandolas en su forma de paridad o no paridad.

Para el diseño del automata se decidio leer toda una cadena completa y devolver un booleano, si el automata concluyo en un estado exitoso, ademas de ser implementado con condicionales

```
1 def parity_automaton(string):
2     state = 'q0'
3     for bit in string:
4         if bit == '0' and state == 'q0':
5             state = 'q1'
6         elif bit == '1' and state == 'q0':
7             state = 'q3'
8         elif bit == '0' and state == 'q1':
9             state = 'q0'
10        elif bit == '1' and state == 'q3':
11            state = 'q0'
12        elif bit == '1' and state == 'q1':
13            state = 'q2'
14        elif bit == '1' and state == 'q2':
15            state = 'q1'
16        elif bit == '0' and state == 'q2':
17            state = 'q3'
18        else:
19            state = 'q2'
20
21    if state == 'q0':
22        return True
23    else:
```

```
return False
```

Siguiendo el siguiente esquema:

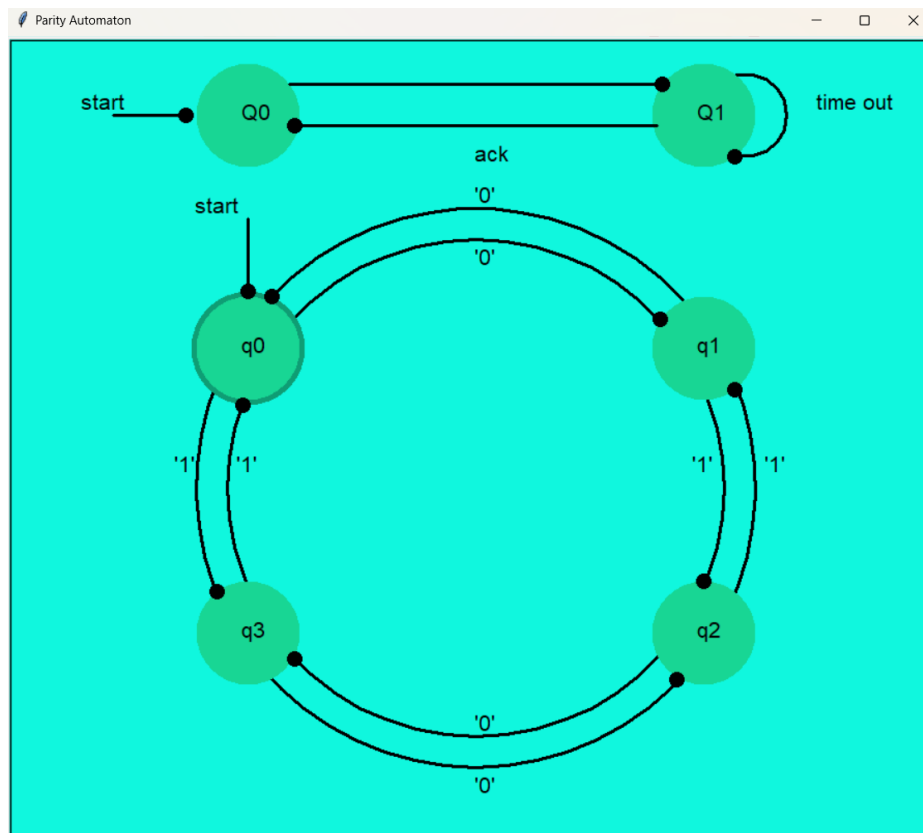


Figura 3.1: Gráfica de automata de Paridad.

Y como estamos hablando de crear un protocolo el cual funcionara de manera infinita, siempre y cuando este activo el autonoma, se opto por tener el codigo principal en un ciclo 'while'

```

1 def main(args):
2     while True:
3
4         if checkAutomatonStatus():
5             bits = args.bits
6             total = args.total
7             createStrins(bits, total)
8             # createStrins(64, 1000000)
9             parity = open('parity.txt', 'w')
10            nonparity = open('nonparity.txt', 'w')
11
12            # Sleep

```

```

13         time.sleep(2)
14
15     binary_strings = open('BinaryStrings.txt', 'r')
16     for line in binary_strings:
17         if parity_automaton(line.strip()):
18             parity.write(line)
19         else:
20             nonparity.write(line)
21
22     # closing files
23     parity.close()
24     nonparity.close()
25     binary_strings.close()
26
27     plotStates()

```

3.0.2. Complejidad

Complejidad Temporal

La complejidad del algoritmo se divide en dos componentes principales: la generación de las cadenas, la lectura de cada bit en las cadenas para poder clasificar como paridad o no paridad

Para la generación de las cadenas, la complejidad temporal es:

$$T : O(k)$$

donde k es el numero de cadenas

Por otro lado, al considerar la función del automata para leer, las cadenas y clasificarlas, las tenemos que leer linealmente todas, teniendo un complejidad de:

$$T : O(bk)$$

donde k es el numero de cadenas y b el numero de bits

En conjunto, la complejidad espacial total del algoritmo es:

$$T : O(bk) + O(k) = O(k)$$

Complejidad Espacial

Dentro del algoritmo, la generación del output no se considera en la complejidad espacial, lo que resulta en una complejidad temporal de:

$$S : O(1)$$

Si tenemos en cuenta que formamos tres archivos la suma de la complejidad espacial sería:

$$S : O(bk) + O(bk) = O(k)$$

3.0.3. Código fuente.

```
1 import random
2 import time
3
4 import turtle
5
6 import argparse
7
8
9 def main(args):
10     bits = 64
11     total = 100000000
12     while True:
13
14         if checkAutomatonStatus():
15             createStrins(bits, total)
16             parity = open('parity.txt', 'a')
17             nonparity = open('nonparity.txt', 'a')
18
19             # Sleep
20             time.sleep(2)
21
22             binary_strings = open('BinaryStrings.txt', 'r')
23             for line in binary_strings:
24                 if parity_automaton(line.strip()):
25                     parity.write(line)
26                 else:
27                     nonparity.write(line)
28
29             # closing files
30             parity.close()
31             nonparity.close()
32             binary_strings.close()
33
34             plotStates()
35
36
37
38
39
40 def initWindow():
41     window = turtle.Screen()
42     window.setup(width=900, height=850)
43     window.bgcolor('#10F7DE')
44     window.title('Parity Automaton')
45     return window
```

```

46
47 def initTurtle():
48     simon = turtle.Turtle()
49     simon.speed('fastest')
50     simon.hideturtle()
51     simon.setpos(0,0)
52     simon.pensize(3)
53     simon.pendown()
54     return simon
55
56 def drawConnectionLines(x, y, radius, simon):
57     simon.penup()
58     simon.setpos(x, y)
59     simon.pendown()
60     simon.circle(radius)
61
62 def drawStates(x, y, simon, state):
63     simon.penup()
64     simon.setpos(x, y)
65     simon.pendown()
66     simon.dot(100, '#18D694')
67     simon.penup()
68
69     simon.setpos(x - 5, y - 10)
70     simon.write(state, font=('Arial', 16, 'normal'))
71
72 def drawArrows(x, y, simon):
73     simon.penup()
74     simon.setpos(x, y)
75     simon.pendown()
76     simon.dot(15, '#000000')
77
78 def writeArrows(x, y, simon, label):
79     simon.penup()
80     simon.color('#000000')
81     simon.setpos(x, y)
82     simon.pendown()
83     simon.write(label, font=('Arial', 16, 'normal'))
84
85
86 def plotStates():
87
88
89     turtle.TurtleScreen._RUNNING=True
90     turtle.tracer(0, 0)
91     simon = initTurtle()
92     window = initWindow()

```

```

93
94
95
96 connection_lines_data = [[0, -250, 240], [0, -280, 270]]
97
98 for pos in connection_lines_data:
99     drawConnectionLines(pos[0], pos[1], pos[2], simon)
100
101
102
103 # Final state, double circle
104 simon.penup()
105 simon.setpos(-220, 125)
106 simon.pendown()
107 simon.dot(110, '#0F9D74')
108 simon.penup()
109
110
111 states = ['q0', 'q1', 'q2', 'q3']
112
113 pos_states = [[-220, 125], [220, 125], [220, -150], [-220, -150]]
114
115 for i in range(len(states)):
116     drawStates(pos_states[i][0], pos_states[i][1], simon, states[i])
117
118 # Start
119 simon.penup()
120 simon.setpos(-220, 250)
121 simon.pendown()
122
123 simon.right(90)
124 simon.forward(70)
125 simon.dot(15, '#000000')
126
127 simon.penup()
128 simon.setpos(-270, 250)
129 simon.write('start', font=('Arial', 16, 'normal'))
130
131
132 # Arrows
133 arrows_pos = [[-197, 175], [-225, 70], [250, 85], [178, 153], [194, -195],
134               [220, -100], [-250, -110], [-175, -175]]
135
136 for pos in arrows_pos:
137     drawArrows(pos[0], pos[1], simon)
138

```

```

139 # Writing arrows
140 arrows_data = [["'0'", 0, 200], ["'0'", 0, 260], ["'0'", 0, -250], ["'0'", 0,
141 -310], ["'1'", -230, 0], ["'1'", -290, 0], ["'1'", 210, 0], ["'1'", 280, 0]]
142
143 for data in arrows_data:
144     writeArrows(data[1], data[2], simon, data[0])
145
146 # Drawing protocol
147 drawStates(-220, 350, simon, 'Q0')
148
149 drawConnectionLines(220, 350, 40, simon)
150 drawStates(220, 350, simon, 'Q1')
151
152 # Lines
153 simon.penup()
154 simon.setpos(-180, 380)
155 simon.pendown()
156 simon.setpos(180, 380)
157 drawArrows(180, 380, simon)
158
159 simon.penup()
160 simon.setpos(175, 340)
161 simon.pendown()
162 simon.setpos(-175, 340)
163 drawArrows(-175, 340, simon)
164
165 drawArrows(250, 310, simon)
166
167 # Start
168 simon.penup()
169 simon.setpos(-350, 350)
170 simon.pendown()
171
172 simon.right(90)
173 simon.right(90)
174 simon.right(90)
175 simon.forward(70)
176 simon.dot(15, '#000000')
177
178 simon.penup()
179 simon.setpos(-380, 350)
180 simon.write('start', font=('Arial', 16, 'normal'))
181
182 simon.penup()
183 simon.setpos(330, 350)
184

```

```

185     simon.write('time out', font=('Arial', 16, 'normal'))
186
187     simon.penup()
188     simon.setpos(0, 300)
189     simon.write('ack', font=('Arial', 16, 'normal'))
190
191
192
193     # Window close
194     window.exitonclick()
195
196
197
198     '''
199         | start
200         v      '0'
201     [[q0]] <--> [[q1]]
202         ^          ^
203         | '1'      | '1'
204         v          v
205     [[q3]] <--> [[q2]]
206             '0'
207
208     '''
209
210
211
212 def checkAutomatonStatus():
213     status = random.randint(0, 1)
214     if status == 0:
215         return True
216     else:
217         return False
218
219 def createStrins(k, total):
220     bits = k
221     output = open('BinaryStrings.txt', 'a')
222
223     number_zeros_left = bits
224
225     max_number = 2 ** bits
226     for j in range(total):
227
228         output.write(bin(
229             random.randint(0, max_number - 1)
230             )[2:].zfill(number_zeros_left) + '\n')
231

```



```

232     output.close()
233
234 def parity_automaton(string):
235     state = 'q0'
236     for bit in string:
237         if bit == '0' and state == 'q0':
238             state = 'q1'
239         elif bit == '1' and state == 'q0':
240             state = 'q3'
241         elif bit == '0' and state == 'q1':
242             state = 'q0'
243         elif bit == '1' and state == 'q3':
244             state = 'q0'
245         elif bit == '1' and state == 'q1':
246             state = 'q2'
247         elif bit == '1' and state == 'q2':
248             state = 'q1'
249         elif bit == '0' and state == 'q2':
250             state = 'q3'
251         else:
252             state = 'q2'
253
254     if state == 'q0':
255         return True
256     else:
257         return False
258
259 def parse_args():
260     # setup arg parser
261     parser = argparse.ArgumentParser()
262
263     # add arguments
264     parser.add_argument("bits",
265                         type=int, help="length of binary strings to be generated",
266                         default=8, nargs='?')
267
268     parser.add_argument("total",
269                         type=int, help = 'total number of strings to be generated',
270                         default=10, nargs='?')
271
272     # parse args
273     args = parser.parse_args()
274
275     # return args
276     return args
277
278 # run script
279 if __name__ == "__main__":

```

```

279 # add space in logs
280 print("\n\n")
281 print("*" * 60)
282 start = time.time()
283
284 # parse args
285 args = parse_args()
286
287 # run main function
288 main(args)
289
290 end = time.time()
291 print("Total time taken: {}s (Wall time)".format(end - start))
292 print("Number of total strings checked: {}".format(args.total))
293 print('Size of each string: {}'.format(args.bits))
294 # add space in logs
295 print("*" * 60)
296 print("\n\n")

```

64 Bits con cien mil cadenas.

- Output en archivo parity.txt:

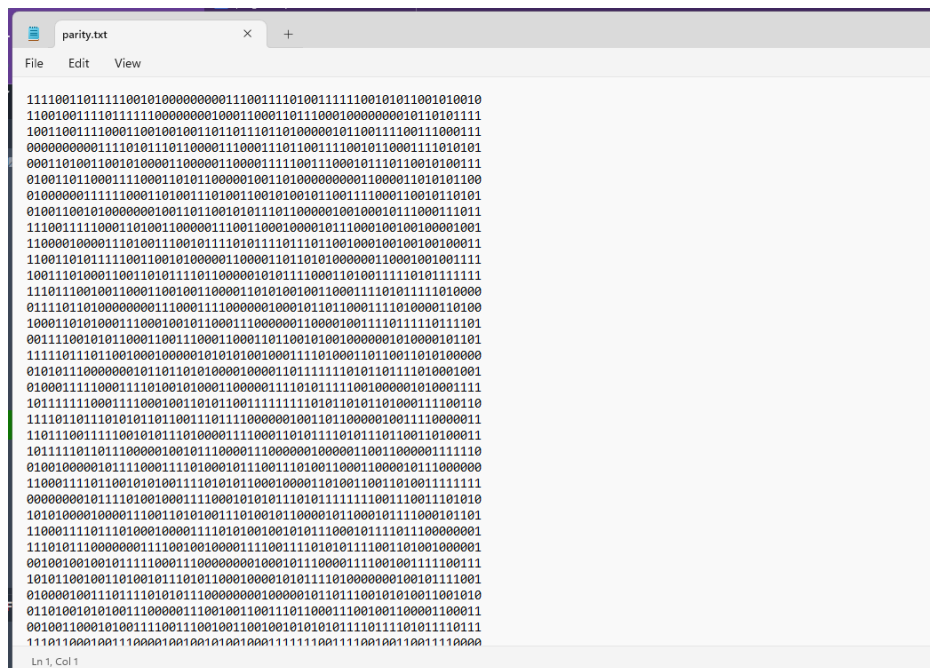
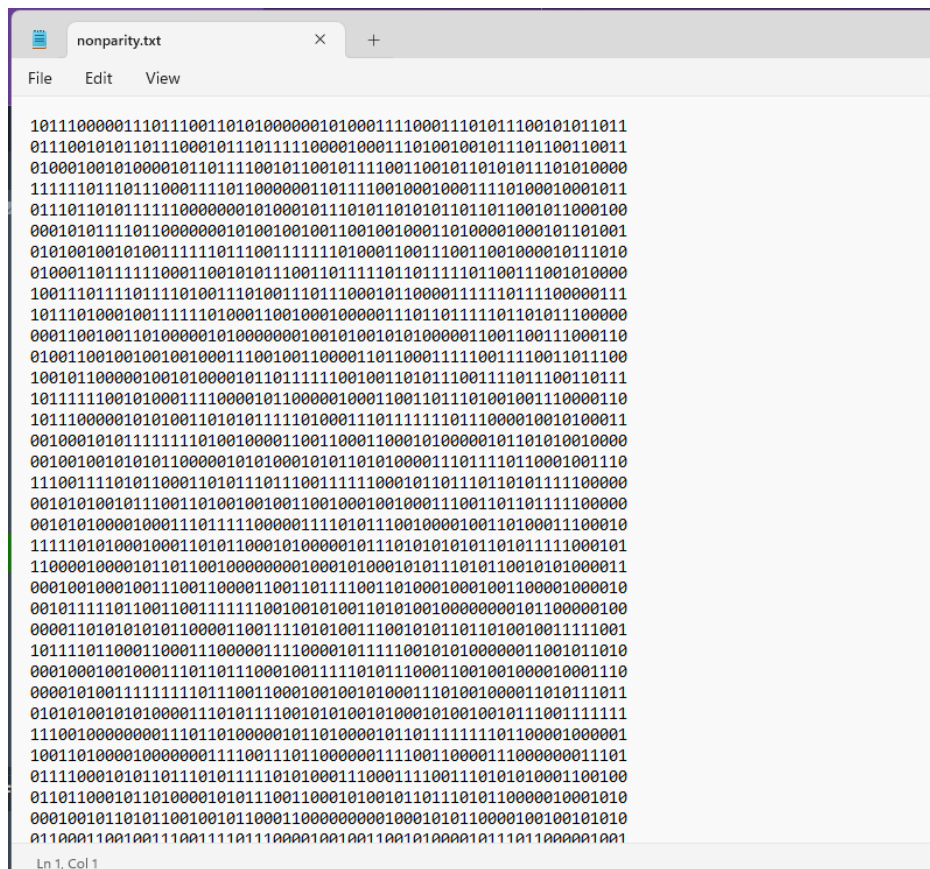


Figura 3.2: Gráfica de automata de Paridad.

- Output en archivo nonparity.txt:



```

10111000001110111001101010000010100011100011101011100101011011
011100101011011100010111011110000100011101001001011101100110011
010001001010000101101110010110010111100110010101011101010000
111111011101110001111011000001101111001000100011110100010001011
011101010111110000001010001011101011010110111011001011000100
000101011101100000001010010010011001001000110100001000101101001
0101001001010011111101110011111101000110011100100100010111010
010001011111000110010101110011011110110111101100111001010000
10011101111011101001110100111011100010110000111111011100000111
101110100010011111010001100100010000011011011110101011100000
0001100100110100000101000000100100101010000011001100111000110
010011001001001001000111001001100001101000111110011110011011100
100101100000100101000010110111110010011010111001111011100110111
101111100101000111100001011000001000110011011101001001110000110
101110000010101010101011111010001110111110111000010010100011
001000101011111110100100001100110001100010100000101101010010000
001001001010111000001010001010110101000011011110110001001110
111001111010110001101011101110011111000101101110101111100000
0010101001011100110100100100110010001001000111001101111100000
001010100000100011101111000001110101110010000100110100011100010
11111010100010001101011000101000001011101010101101011111000101
1100001000010110110010000000100010100010111010110010101000011
00010010001001110011000011001101111001101000100010100001000010
0010111110110011001111111001001001101010010000000101100000100
0000110101010101100001100111101010011100101011011010010011111001
101111011000110001110000011100001011111001010100000011001011010
000100010010001110110111000100111101011100011001001000010001110
00001010011111111101110011000100100100011101001000011010111011
010101001010100001110101110010101001010001010010010111001111111
1110010000000011101101000001011010000101111111101100001000001
1001101000010000001111001110110000011100110000111000000011101
011110001010110111010111101010001110001111001110101010001100100
0110110001011010000101011100110001010010110111010110000010001010
00010010110101100100110001100000000100010110000100100101010
0110001100100111001111011100001001001100001001001101100001001

```

Figura 3.3: Gráfica de automata de Paridad.

- Output de todas las cadenas .txt:

The screenshot shows a text editor window titled 'BinaryStrings.txt'. The editor contains a single line of text consisting of a long binary string. The string is composed of 0s and 1s, representing the output of a parity automaton. The status bar at the bottom indicates 'Ln 10, Col 20'.

```

111100110111110010100000000110011110100111110010101001010010
110010011110111110000000010001100011011100010000000101010111
1001100111100011001001001101101110100000101100111100111000111
101110000011101100110101000000100001110001110101100101011011
00000000011101011101000011000111010011110010110001111010101
00011010011001010000110000110000111100111000101110110010100111
010011011000111100011010110000010011010000000011000011010101100
01110010101011100010111011110000100011101001001011101100110011
0100000011111000110100111010011001011001111000110010110101
010001001010000101101110010110010111001100101101011101010000
0100110010100000010011011001011101100000100100010111000111011
111001111100011010011000001100110001000010111000100100001001
110000100001110100111001011110101110111011001000100100100011
11111011101110001111011000000110111001000100011110100010001011
1100110101111100110010100000110000110110100000110001001001111
100111000110011101011101100000101111000110100111110101111111
11101110010011000110010011000011010010011000111101011111010000
011110110100000001110001111000000100010110110001111010000110100
100011010100011100010010110001110000001100001001111011110111101
0111011010111110000000101000101110101101011011011001011000100
0001010111101100000001010010011001001000110100001000101101001
010100100100111110111001111110100011001110011001000010111010
0011110010101100011001110001100011011001010010000001010000101101
01000110111111000110010101110011011110110111101100111001010000
11111011101100100010000010101001000111101000110110011010100000
01010111000000010110110101000100001101111110101101111010001001
100111011110111101001110100111011100010110000111110111100000111
01000111110001111010011000110000011110101111001000001010001111
10111010001001111110100011001000100000110110111110110101110000
1011111110001111000100110101100111111111010110101000111100110
1111010111010101101110111100000010011010000010011110000011
0001100100110100000101000000010010100101010000011001100111000110
1101110011111001010111010000111100011010111101101100110100011
1011111011011100000100101110000111000000100000110011000001111110
0100110000010111100011110001111000111100011100011100000000000000

```

Figura 3.4: Gráfica de automata de Paridad.

- Gráfica del Automata:

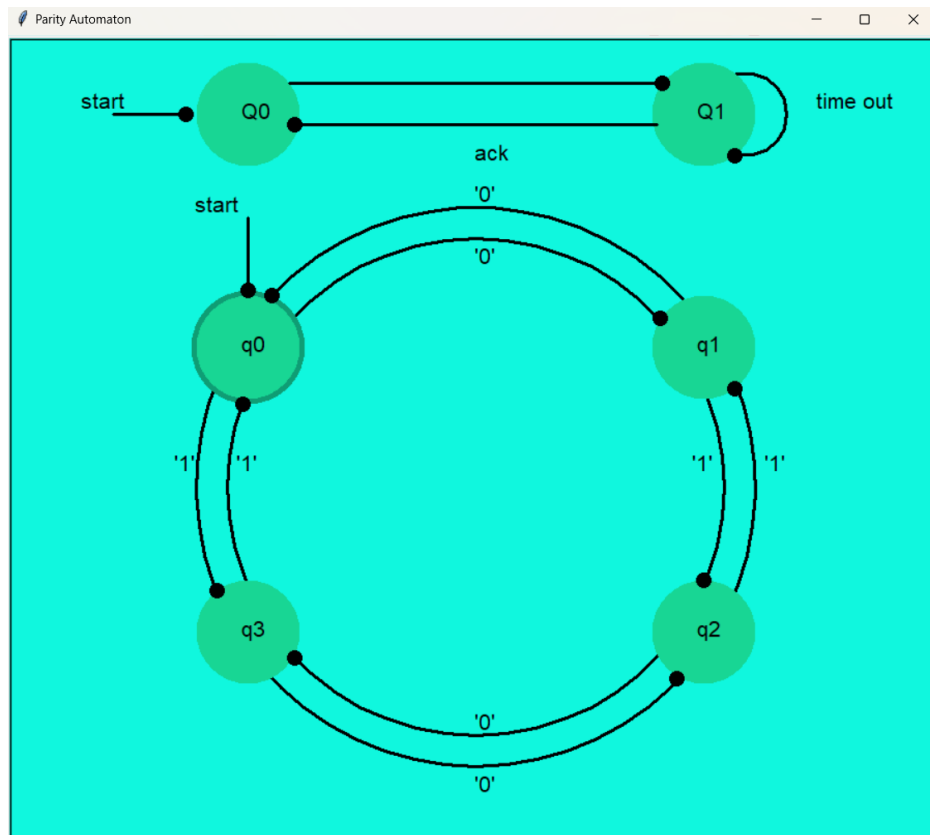


Figura 3.5: Gráfica de automata de Paridad.

Conclusión.

Crear un autómata determinista puede resultar muy fácil de implementar en código, ya que cada estado del autómata está reflejado en el esquema y el mismo autómata no puede estar en dos estados a la vez. Sin embargo, a pesar de que existe una mayor eficiencia y es más fácil resolver problemas reales y plasmarlos en un autómata no determinista, es posible tener un autómata determinista equivalente para cada no determinista.

La aplicación de un autómata determinista es más sencilla cuando se trata de implementar un protocolo, siempre y cuando esté activo.

La creación del protocolo con la unión del autómata de paridad nos facilita muchas cosas a la hora de programarlo, ya que es muy sencillo y no requiere almacenamiento adicional en el equipo de cómputo, en lugar de usar un contador para la paridad u otra estructura de datos.

Referencias.

- Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). Introduction to Automata Theory, Languages, and Computation (3rd ed.). Pearson.
- Mike (2020, febrero 02). El Patrón de los Números Primos y la Hipótesis de Riemann [Archivo de video]. Mates Mike. <https://www.youtube.com/watch?v=cZJv2FKutPU>