UNIVERSIDAD
Finis Terrae

Universidad Finis Terrae
FACULTAD DE INGENIERÍA

Conversión de Expresiones Regulares a Autómatas Finito No Determinista (AFND) y Autómata Finito Determinista (AFD)

Autores:
Nicolas Arellano
Miguel Cornejo
Benjamin Sepulveda L

Asignatura:
Teoría de la Computación

Profesor guía:
Rodrigo Andres Paredes Moraleda

Fecha de entrega:
24/11/2024

NRC:
78093

Índice

1. Introducción:

Este proyecto integra el desarrollo de un programa en Python y una interfaz gráfica interactiva para la conversión de expresiones regulares (ER) en autómatas finitos no determinísticos (AFND) utilizando el método de Thomson, y la posterior determinización de estos en autómatas finitos determinísticos (AFD). Estos procesos son fundamentales en el ámbito de la computación, ya que permiten construir autómatas que sirven como base para diversos lenguajes de programación, sistemas de validación y herramientas de procesamiento de texto.

El programa no solo realiza las conversiones matemáticas necesarias, sino que también incorpora una visualización gráfica dinámica de los autómatas generados, permitiendo una mejor comprensión de sus estructuras y transiciones. Además, incluye la funcionalidad de detección de patrones en texto, reportando las posiciones donde las ocurrencias de las ER terminan, unificando así el rigor computacional con una experiencia de usuario accesible e intuitiva.

La interfaz gráfica, se desarolla principalmente con la herramienta *customtkinter*, facilita la interacción del usuario al permitirle introducir expresiones regulares, visualizar los autómatas generados en forma de grafos y simular la detección de patrones en tiempo real. Esta combinación de funcionalidades asegura que tanto el componente técnico como el visual estén integrados, ofreciendo una herramienta completa para el análisis y la enseñanza de autómatas finitos.

Mediante técnicas como el método de Thomson para la construcción del AFND y algoritmos de determinización para convertir el AFND en un AFD, el programa maneja correctamente operaciones fundamentales como concatenación, unión y clausura de Kleene. Este informe documenta no solo el diseño y desarrollo del programa, sino también las decisiones técnicas, los desafíos enfrentados y las pruebas realizadas, destacando cómo se combinaron las bases matemáticas con el diseño interactivo para producir un sistema funcional y didáctico.

2. Análisis del Problema

La conversión de expresiones regulares a autómatas es un proceso complejo que implica transformar una secuencia de símbolos y operadores en una estructura de estados y transiciones que represente adecuadamente el patrón buscado. Los autómatas finitos, tanto determinísticos como no determinísticos, ofrecen un marco robusto para representar patrones definidos por expresiones regulares.

El problema central que se aborda es la detección eficiente de patrones en un texto con alfabeto de letras mayúsculas y minúsculas, utilizando expresiones regulares y autómatas. Para resolver este problema, es necesario transformar una expresión regular en una estructura que pueda reconocer el patrón en cualquier posición del texto de manera óptima. Esta transformación implica tres etapas fundamentales:

- **Conversión de ER a AFND:** Las ER proporcionan una forma de describir patrones mediante la concatenación, unión y clausura de Kleene. Empleando el método de Thomson, cada símbolo y operador de la ER se traduce en transiciones y estados, generando un autómata que representa fielmente la expresión.

- **Determinización del AFND para obtener un AFD:** Los autómatas no deterministas permiten manejar múltiples transiciones desde un mismo estado, lo cual puede ser ineficiente en sistemas prácticos. La determinización es importante para obtener un autómata en el que cada estado tiene un único destino para cada símbolo, eliminando la ambigüedad y simplificando el reconocimiento de patrones.

- **Eficiencia y simplicidad:** La implementación debe ser lo suficientemente modular y clara para facilitar la integración de las fases posteriores (visualización del programa y detección de patrones en texto).

- **Validación de entradas:** Dado que el sistema no admite caracteres especiales, es importante que el programa pueda identificar y rechazar expresiones regulares no válidas.

Desafíos técnicos

- **Construcción del AFND**: El método de Thomson debe aplicarse correctamente para garantizar que todas las operaciones de concatenación, unión y estrella de Kleene se traduzcan en transiciones adecuadas.
- **Determinización**: La conversión de AFND a AFD puede ser computacionalmente costosa, ya que implica el manejo de subconjuntos de estados que aumentan exponencialmente. Para

optimizar el proceso, es esencial calcular las clausuras epsilon de manera eficiente y estructurar las transiciones de modo que el autómata resultante sea lo más simple posible.
- **Procesamiento del texto:** Debe realizarse de manera eficiente, garantizando que el tiempo de ejecución sea manejable incluso con textos largos y patrones complejos.

3. Diseño de la solución

La solución desarrollada está compuesta por dos clases principales del programa: *Thomson* y *Conversion*, cada una encargada de una fase clave del proyecto. La clase *Thomson* se utiliza para realizar la conversión de la ER en un AFND, mientras que la clase *Conversion* se encarga de la determinización del AFND para obtener un AFD. En conjunto, ambas clases implementan los algoritmos de procesamiento y generación de autómatas que cumplen con las reglas de la teoría de autómatas.

3.1 Conversión de ER a AFND

Para la conversión de una Expresión Regular (ER) en un Autómata Finito No Determinista (AFND), se utiliza el método de Thomson. Este método es adecuado para la construcción de autómatas debido a su estructura modular, donde cada símbolo y operador en la ER se convierte en un conjunto específico de estados y transiciones, representando cada posible camino de reconocimiento del patrón.

3.1.1 Estructura de la Clase Thomson

Se diseñó una clase denominada *Thomson* que contiene métodos especializados para crear y organizar estados y transiciones de acuerdo con los operadores encontrados en la ER:

- Método *crear_estado*: Este método crea y devuelve un nuevo estado único. Cada estado se nombra secuencialmente (por ejemplo, *q0, q1,* etc.), lo que permite una fácil identificación y seguimiento.

- Método *simplificacion_er*: Este método verifica y simplifica la ER, asegurando que los operadores redundantes no se dupliquen y que la expresión esté en una forma apta para su procesamiento.

- Método *AFND*: Este método recorre la ER y, basándose en el método de Thomson, crea transiciones específicas para cada operador:

El método de Thomson se basa en tres operaciones principales para construir el AFND:

- **Concatenación (.):** Este operador establece una secuencia en la que deben aparecer los símbolos, uno tras otro. Para su representación, el estado final de la primera secuencia se conecta al estado inicial de la segunda, creando un flujo secuencial que refleja la concatenación de dos símbolos o subexpresiones en la ER.

- **Unión (|)**: Permite que el autómata elija entre dos caminos distintos, cada uno representando una opción alternativa. Para la unión, se agrega un nuevo estado inicial con transiciones ε (epsilon) hacia los estados iniciales de cada alternativa en la expresión. Los estados finales de cada ruta convergen en un estado final común, garantizando que cualquiera de los dos caminos sea una coincidencia válida.

- **Estrella de Kleene (*)**: Este operador representa cero o más repeticiones de una secuencia. Para implementar la clausura de Kleene, se crean transiciones ε que permiten al autómata repetirse sobre sí mismo sin consumir ningún símbolo, así como una transición hacia un estado final que permite que la secuencia sea opcional (es decir, puede aparecer cero veces).

3.2 Conversión de AFND a AFD

Para el programa se crea una clase ***Conversion*** implementa el proceso de determinización, es decir, convierte el AFND en un AFD. Este paso es crucial para obtener un autómata en el que cada estado y cada símbolo de entrada tengan un único destino, eliminando la indeterminación del AFND.

Estructura de la Clase Conversión:

- Método clausuras: Calcula la clausura epsilon de cada estado del AFND, que representa el conjunto de estados accesibles mediante transiciones epsilon (transiciones que no consumen ningún símbolo).

- Método AFD: Utiliza las clausuras epsilon calculadas para construir transiciones determinísticas. Este método asegura que cada estado en el AFD tenga una única transición por cada símbolo del diccionario de entrada.

A partir de un AFND que contiene transiciones con epsilon y múltiples caminos posibles, el método de clausuras epsilon crea grupos de estados alcanzables desde un solo estado inicial, consolidando estos conjuntos en nuevos estados determinísticos en el AFD.

4 Requisitos del Sistema

Para la correcta ejecución de este programa, se recomienda cumplir con los siguientes requisitos:

- Sistema Operativo: Windows, macOS o Linux.
- Python: Versión 3.7 o superior, instalada
- (Opcional) Visual Studio Code con la Extensión de Python incorporada

5. Código Base (Sin Interfaz Gráfica):

Se presenta el siguiente código del funcionamiento del proyecto para las expresiones regulares, y sus transformaciones autómatas:

```
1   ##Code made by Nicolás Arellano
2   class Thomson:
3       def __init__(self):
4           self.K = []  # Lista de estados
5           self.delta = []  # Lista de transiciones
6           self.lista_er = []  # Lista de la expresión regular
7           self.estado = 0  # Contador para los estados
8           self.primera_iteracion = True
9           self.cadena = []
10          self.diccionario = []
11  }
12      def juntar(self, er):
13          agregar = "-"
14          er2 = ""
15          while "-" in agregar:
16              agregar = input("Ingresar continuación (Expresión Regular): ")
17              er += agregar
18
19          for i in range(len(er)):
20              if er[i] != "-":
21                  er2 += er[i]
22
23          return er2
24
25      def juntar_cadena(self, cadena):
26          agregar = "-"
27          cadena2 = ""
28          while "-" in agregar:
29              agregar = input("Ingresar continuación (Cadena): ")
30              cadena += agregar
31
32          for i in range(len(cadena)):
33              if cadena[i] != "-":
34                  cadena2 += cadena[i]
35
36          return cadena2
37
38
39      def crear_estado(self):
```

```
40          if self.primera_iteracion == True and self.estado != 0:
41              self.estado -= 1
42              estado = "q" + str(self.estado)
43              self.primera_iteracion = False
44              self.estado += 1
45              return estado
46          elif self.estado == 0:
47              estado = "q" + str(self.estado)
48              self.estado += 1
49              self.primera_iteracion = False
50              return estado
51          else:
52              estado = "q" + str(self.estado)
53              self.estado += 1
54              return estado
55
56      def agregar_estado(self, *estados):
57          for estado in estados:
58              if estado not in self.K:
59                  self.K.append(estado)
60
61      def simplificacion_er(self, er):
62          resultado = ""
63          pila = []
64          i = 0
65
66          # Reglas para eliminar falencias y simplificar
67          while i < len(er):
68              char = er[i]
69
70              # Ignorar caracteres no permitidos
71              if char not in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ*|.0Φ_":
72                  print(f"Carácter inválido '{char}' eliminado.")
73                  i += 1
74                  continue
75
76              # Manejar paréntesis, aunque no están permitidos en este caso
77              if char in "([{)]}":
78                  print(f"Paréntesis o corchetes no permitidos '{char}' eliminado.")
79                  i += 1
80                  continue
81
82              # Eliminar operadores consecutivos inválidos
83              if char in "|*.":
```

```
84              if resultado and resultado[-1] in "|*.":
85                  print(f"Operador consecutivo '{resultado[-1]} {char}' corregido a '{char}'.")
86                  resultado = resultado[:-1] + char
87              elif not resultado or resultado[-1] in "|*.":
88                  print(f"Operador mal posicionado '{char}' eliminado.")
89              else:
90                  resultado += char
91
92          # Clausura de Kleene (*) correctamente usada
93          elif char == "*" and (not resultado or resultado[-1] in "|*."):
94              print(f"Clausura de Kleene mal usada '{char}' eliminada.")
95          else:
96              resultado += char
97
98          i += 1
99
100     # Eliminar '|' o '.' al final
101     if resultado.endswith("|") or resultado.endswith("."):
102         print(f"Operador '{resultado[-1]}' al final eliminado.")
103         resultado = resultado[:-1]
104
105     # Eliminar expresiones vacías resultantes
106     if not resultado:
107         print("Expresión regular vacía tras simplificación.")
108         return "Φ"
109
110     print(f"Expresión regular simplificada: {resultado}")
111     return resultado
112
113
114 def AFND(self, er):
115     # Convertir la expresión regular en una lista de caracteres
116     if "-" in er:
117         er = self.juntar(er)
118     #creacion del primero estado que acepta todos los caracteres de la cadena
119     q_inicio = self.crear_estado()
120     for simbolo in self.diccionario: # hace que todos los simmbolos del alfabeto vayan para ahi cosa que no se
121 pueda salir, por algo sumidero lol
122         self.delta.append([q_inicio, simbolo, q_inicio])
123     self.primera_iteracion = True
124
125     er = self.simplificacion_er(er)
126
127
```

```
128         #creacion de los estados normalmente
129         for i in er:
130             self.lista_er.append(i)
131         i = 0
132         while i < len(self.lista_er):
133             # Si no es Φ o 0, procesamos normalmente
134             if self.lista_er[i] != "Φ" and self.lista_er[i] != "0":
135                 if i+1 < len(self.lista_er) and self.lista_er[i+1] == "*":
136                     # Como es clausura de Kleene, siempre se añaden 4 estados
137                     q0 = self.crear_estado()
138                     q1 = self.crear_estado()
139                     q2 = self.crear_estado()
140                     q3 = self.crear_estado()
141 
142                     # Añadir las transiciones (deltas)
143                     self.delta.append([q0, "_", q1])
144                     self.delta.append([q1, self.lista_er[i], q2])
145                     self.delta.append([q2, "_", q1])
146                     self.delta.append([q2, "_", q3])
147                     self.delta.append([q0, "_", q3])
148 
149                     # Añadir los nuevos estados
150                     self.agregar_estado(q0, q1, q2, q3)
151 
152                     # Saltar al siguiente símbolo (ya que el actual era parte de "*")
153                     i += 2
154                     self.primera_iteracion = True
155                 ##si detecto un epsilon:
156                 elif i == "_":
157                     q0 = self.crear_estado()
158                     q1 = self.crear_estado()
159 
160                     self.delta.append([q0, "_", q1])
161                     self.agregar_estado(q0, q1)
162 
163                     i += 1
164                     self.primera_iteracion = True
165                 elif i+3 < len(self.lista_er) and self.lista_er[i+1] == "|" and self.lista_er[i+3] == "*":
166                     q0 = self.crear_estado()
167                     q1 = self.crear_estado()
168                     q2 = self.crear_estado()
169                     q3 = self.crear_estado()
170                     q4 = self.crear_estado()
171                     q5 = self.crear_estado()
```

```
172             q6 = self.crear_estado()
173             q7 = self.crear_estado()
174
175
176             self.delta.append([q0, "_", q1])  # Epsilon desde el inicio a la rama 'a'
177             self.delta.append([q1, self.lista_er[i], q2])
178             self.delta.append([q2, "_", q7])
179             self.delta.append([q0, "_", q3])
180             self.delta.append([q3, "_", q4])
181             self.delta.append([q4, self.lista_er[i+2], q5])
182             self.delta.append([q5, "_", q4])
183             self.delta.append([q5, "_", q6])
184             self.delta.append([q3, "_", q6])
185             self.delta.append([q6, "_", q7])
186
187             # Agregar todos los estados
188             self.agregar_estado(q0, q1, q2, q3, q4, q5, q6, q7)
189             i += 4
190             self.primera_iteracion = True
191
192
193         elif i+1 < len(self.lista_er) and self.lista_er[i+1] == "|":
194             q0 = self.crear_estado()
195             q1 = self.crear_estado()
196             q2 = self.crear_estado()
197             q3 = self.crear_estado()
198             q4 = self.crear_estado()
199             q5 = self.crear_estado()
200
201             self.delta.append([q0, "_", q1])
202             self.delta.append([q1, self.lista_er[i], q2])
203             self.delta.append([q2, "_", q5])
204             self.delta.append([q0, "_", q3])
205             self.delta.append([q3, self.lista_er[i+2], q4])
206             self.delta.append([q4, "_",q5])
207
208             # Añadir los nuevos estados
209             self.agregar_estado(q0, q1, q2, q3, q4, q5)
210
211             # Saltar al sub-siguiente símbolo (ya que el actual era parte del or y el siguiente a ese)
212             if i+3 < len(self.lista_er) and self.lista_er[i+3] == "|":
213                 i += 2
214             else:
215                 i += 3
```

```
216                 self.primera_iteracion = True
217             else:
218                 q0 = self.crear_estado()
219                 q1 = self.crear_estado()
220 
221                 self.delta.append([q0, self.lista_er[i], q1])
222                 self.agregar_estado(q0, q1)
223                 if i+1 < len(self.lista_er) and self.lista_er[i+1] == ".":
224                     i += 2
225                 else:
226                     i += 1
227                 self.primera_iteracion = True
228 
229         # sumidero
230         else:
231             q_sumidero = self.crear_estado()
232             for simbolo in self.diccionario: # hace que todos los simmbolos del alfabeto vayan para ahi cosa que no
233 se pueda salir, por algo sumidero lol
234                 self.delta.append([q_sumidero, simbolo, q_sumidero])
235 
236             i = len(self.lista_er)
237 
238     def sigma(self,cadena):
239         diccionario = []
240         if "-" in cadena:
241             cadena = self.juntar_cadena(cadena)
242         # Asegurarse de que delta no esté vacío y que cada elemento tenga al menos 3 elementos
243         for i in cadena:
244             if i not in diccionario:
245                 diccionario.append(i)
246         self.diccionario = diccionario
247 
248 class Conversion:
249     def __init__(self):
250         self.clausura_estado = []  # hasta donde puede llegar con y solo transiciones epsilon
251         self.diccionario = []
252         self.delta_min = []
253         self.estados_finales = []
254         self.estados_totales = []
255 
256     def clausuras(self, estados, transiciones):
257         epsilon_encontrado = False  # Variable para verificar si alguna vez encontramos epsilon
258         for m in range(len(estados)):
259             epsilon = False
```

```
260             clausura_temporal = [estados[m], "U"]  # Inicializamos con el estado y "U"
261
262             for n in range(len(transiciones)):
263                 if estados[m] == transiciones[n][0] and transiciones[n][1] == "_":
264                     epsilon_encontrado = True
265                     epsilon = True
266                     state = transiciones[n][2]
267                     clausura_temporal.append(state)
268
269             # Si se encontraron transiciones epsilon, usamos la clausura completa.
270             # Si no, dejamos solo el estado inicial sin duplicados ni "U".
271             if epsilon:
272                 self.clausura_estado.append(clausura_temporal)
273             else:
274                 self.clausura_estado.append([estados[m]])
275
276         ####print("Clausuras: ", self.clausura_estado)  # Imprimimos las clausuras generadas
277
278         self.AFD(transiciones, estados)  # Llama al objeto AFD para que comience el cambio de AFND a AFD
279
280
281
282     def AFD(self, transiciones, estados):
283         # Inicializamos con la primera clausura sin 'U'
284         estado_inicial = [i for i in self.clausura_estado[0] if i != "U"]
285         estados_2 = [estado_inicial]
286         diccionario = p.diccionario
287         trans = transiciones
288         procesados = set()
289         self.delta_min = []
290
291         while estados_2:
292             estado_actual = estados_2.pop(0)
293             estado_actual_tuple = tuple(estado_actual)  # Convertimos a tupla para usar en el set
294             if estado_actual_tuple in procesados:
295                 continue
296             procesados.add(estado_actual_tuple)
297
298             for c in diccionario:
299                 t = set()
300                 for estado in estado_actual:
301                     for transicion in trans:
302                         if estado == transicion[0] and transicion[1] == c:
303                             destino = transicion[2]
```

```
304                         for clausura in self.clausura_estado:
305                             if clausura[0] == destino:
306                                 t.update([s for s in clausura if s != "U"])
307 
308                 if not t:
309                     t = ['Φ']
310                 else:
311                     t = list(t)  # Convertimos `t` a lista en lugar de tupla para mantener el formato original
312 
313                 # Agregamos la transición a delta_min con formato de lista
314                 self.delta_min.append([estado_actual, c, t])
315 
316                 # Si el estado destino no ha sido procesado ni está en la lista, lo agregamos
317                 if t != ['Φ'] and tuple(t) not in procesados and t not in estados_2:
318                     estados_2.append(t)
319         #estados iniciales, finales y totales
320         for i in range(len(self.delta_min)):
321             for j in range(len(self.delta_min[i])):
322                 for b in range(len(self.delta_min[i][j])):
323                     if self.delta_min[i][j][b] == estados[-1] and self.delta_min[i][j] not in self.estados_finales and
324 self.delta_min[i][j][b] not in diccionario:
325                         self.estados_finales.append(self.delta_min[i][j])
326                     if self.delta_min[i][j] not in self.estados_totales and self.delta_min[i][j][b] not in diccionario:
327                         self.estados_totales.append(self.delta_min[i][j])
328 
329 
330 
331 # Modo de uso
332 expresion_regular = input("Ingrese su expresión regular: ")
333 string = input("Ingrese una Cadena: ")
334 
335 if "ñ" in expresion_regular or "Ñ" in expresion_regular:
336     print("Expresion regular inválida: contiene 'ñ' o 'Ñ'")
337 else:
338     p = Thomson()
339     p.sigma(string)
340     p.AFND(expresion_regular)
341     if expresion_regular not in ["0", "Φ", "0Φ", "Φ0","*","*|","|*", "|", "."] and string != "":
342         if len(expresion_regular) <= 50:
343             print("AFND")
344             print("Estado Inicial", p.K[0])
345             print("Estado final", p.K[-1])
346             print("Estados:", p.K)
347             print("Transiciones:", p.delta)
```

```
348             print("Diccionario", p.diccionario)
349
350             print("\t")
351             b = Conversion()
352             b.clausuras(p.K, p.delta)
353             print("AFD")
354             print("Clausulas Epsilon", b.clausura_estado)
355             print("Estado Inicial", p.K[0])
356             print("Estados Finales", b.estados_finales)
357             print("Estados Totales", b.estados_totales)
358             print("Transiciones:", b.delta_min)
359             print("Diccionario", p.diccionario)
360         else:
361             print("Expresion Regular tiene más de 50 caracteres")
362     else:
363         print("Expresion regular vacia o cadena invalida")
```

Figura 1: "Python, Código del programa de ejecución del proyecto"

La clase *Thomson* construye un autómata finito no determinista (AFND) usando el método de Thompson para transformar una expresión regular (ER). Inicialmente, se definen atributos para almacenar estados (*K*), transiciones (*delta*), y caracteres de la ER (*lista_er*). Métodos como juntar y *juntar_cadena* permiten al usuario ingresar una ER o cadena en varias líneas, concatenándolas en una sola. Para cada nuevo estado, *crear_estado* asigna identificadores únicos (*q0, q1*), mientras que *agregar_estado* evita duplicados en la lista de estados.

El método *simplificacion_er* limpia la ER eliminando caracteres innecesarios, asegurando que no haya operadores sueltos y manteniendo solo símbolos válidos. En AFND, se implementa el método de Thompson: se crea un estado inicial (*q_inicio*) que acepta cualquier símbolo del alfabeto. Dependiendo de los operadores de la ER, se construyen rutas con concatenación, unión (|), y clausura de Kleene (*). Se incluye un estado sumidero que maneja cualquier símbolo no válido.

La construcción del alfabeto (*sigma*) se basa en los caracteres únicos de la cadena de entrada, almacenados en diccionario para estandarizar los símbolos aceptados por el autómata. La clase *Conversion* convierte el AFND a un autómata finito determinista (AFD) mediante las clausuras epsilon, calculadas en clausuras para identificar estados alcanzables sin consumir símbolos. Con esta base, AFD crea transiciones para cada símbolo del diccionario y define un estado sumidero (Φ) en caso de transiciones no válidas, completando así el AFD con sus estados finales y transiciones mínimas.

6. Métodos/funciones implementados

1. ***__init__ (Thomson)*:** Inicializa los atributos esenciales para la construcción del autómata, como la lista de estados *K*, transiciones delta, caracteres de la ER en *lista_er*, y el contador estado para los nombres de los estados.

2. **juntar y *juntar_cadena*:** Permiten al usuario ingresar expresiones regulares o cadenas largas en varias líneas, concatenando el texto para construir la entrada completa en una sola línea.

3. ***crear_estado*:** Genera un nuevo estado con el formato *qX* (por ejemplo, *q0*, *q1*), garantizando nombres únicos mediante el uso del contador estado.

4. ***agregar_estado*:** Agrega uno o más estados a la lista *K* sin duplicarlos, manteniendo el conjunto de estados del autómata.

5. ***simplificacion_er*:** Limpia y ajusta la expresión regular eliminando operadores o caracteres innecesarios, evitando duplicados y manteniendo solo símbolos válidos.

6. *AFND*: Construye el autómata finito no determinista (AFND) utilizando el método de Thompson. Crea transiciones para concatenación, unión (|), y clausura de Kleene (*), estableciendo estados iniciales, intermedios y finales para representar la ER.

7. ***sigma*:** Define el alfabeto del autómata a partir de los caracteres únicos en la cadena de entrada, almacenándolos en el diccionario para uso en el AFND y AFD.

8. ***__init__ (Conversion)*:** Inicializa las estructuras necesarias para la conversión del AFND a AFD, incluyendo las clausuras epsilon (*clausura_estado*), las transiciones mínimas (*delta_min*), y los estados finales y totales.

9. **Cláusulas:** Calcula las clausuras epsilon para cada estado en el AFND, identificando los estados alcanzables sin consumir símbolos, lo cual es crucial para construir el AFD.

10. ***AFD*:** Convierte el AFND en un autómata finito determinista (AFD) usando las clausuras epsilon. Genera transiciones para cada símbolo en el diccionario, define un estado sumidero (Φ) en caso de transiciones no válidas, y completa el AFD con sus estados finales y transiciones mínimas.

6.1 Proceso de Expresión Regular (ER) a un Autómata Finito No Determinista (AFND):

El proyecto implementa la conversión de una expresión regular (ER) en un autómata finito no determinista (AFND). Este proceso consiste en descomponer la ER en una serie de símbolos y operadores y construir el AFND mediante estados y transiciones que reflejan la estructura de la expresión. A continuación, se detallan los pasos de conversión.

1. Simplificación de la Expresión Regular

Antes de comenzar la construcción del AFND, el código realiza una simplificación de la ER para eliminar operadores redundantes, símbolos que no están en el texto a procesar y operadores mal colocados, como múltiples operadores (|) consecutivos o finales. También se asegura de que la expresión regular cumpla con las reglas básicas, evitando así errores durante la conversión.

2. Inicialización de Estados y Simbolización

- Se inicializa un conjunto de estados que representarán los nodos en el AFND.
- Se asegura que el AFND pueda consumir todos los caracteres de la cadena de entrada. Se añade un estado inicial con transiciones para cada símbolo presente en la cadena, de modo que el autómata no se quede sin reconocer algún símbolo del alfabeto, esto es conocido como lazo.

3. Procesamiento de Operadores de la ER

La conversión procesa la ER símbolo por símbolo, manejando cada uno de los operadores estándar de la siguiente manera:

- Concatenación (.): Si la ER contiene una concatenación explícita o implícita (o sea que sea del estilo ab y no a.b), se crean dos estados con una transición directa que consume el símbolo. Este operador establece una secuencia de estados donde cada símbolo de la ER tiene un estado sucesor.
- Unión (|): Si se detecta un operador de unión, el código construye una estructura en la que dos ramas pueden conducir a diferentes transiciones desde un estado inicial común hasta un estado final compartido. Esto se realiza mediante:
    - Un estado inicial con transiciones épsilon a dos subautómatas que representan cada alternativa.
    - Los estados finales de cada rama se conectan a un estado final común mediante transiciones épsilon.

- Clausura de Kleene (*): Cuando se encuentra el operador *, se construye un bucle de transición que permite realizar cero o más repeticiones del símbolo precedente. Este operador se representa de la siguiente manera:
    - Un nuevo estado inicial que apunta mediante una transición épsilon al primer estado del subautómata que representa el símbolo o subexpresión bajo *.
    - Transiciones épsilon desde el último estado del subautómata de repetición hacia el estado inicial, formando el bucle, y hacia un estado final, permitiendo que el símbolo pueda repetirse o no aparecer.

4. Creación de Transiciones y Conexión de Estados

A medida que se procesan los operadores, se crean transiciones específicas entre estados. Cada transición contiene:

- Estado de Origen: El estado inicial de la transición.
- Símbolo: El o los símbolos que se deben consumir para que se active la transición. En el caso de transiciones épsilon, el símbolo es utilizado es "_".
- Estado de Destino: El estado al que se transita después de consumir el símbolo.

Quedando de la forma [q_n, símbolo, q_{n+k}]

Con n y k siendo cualquier número real mayor o igual a 0

Estas transiciones se almacenan en la lista delta del AFND.

5. Generación de Estados Inicial y Final

- Estado Inicial: El estado inicial del AFND se establece como el primer estado creado en la construcción, ya que representa el punto de partida de la ER.
- Estado Final: El estado final del AFND se establece en el último estado creado o en el estado de convergencia de transiciones si existen operadores de unión o clausura de Kleene.

6.2 Conversión del Autómata Finito No Determinista (AFND) a un Autómata Finito Determinista (AFD)

A continuación, se explicará cómo el código convierte el AFND generado por la Expresión Regular a un AFD.

1. Creación de Clausuras Epsilon

Para cada estado del AFND, se calcula la **clausura épsilon**, que es el conjunto de todos los estados accesibles mediante transiciones épsilon ("_") sin consumir símbolos del alfabeto. Para construir estas clausuras:

- Se itera sobre cada estado en el conjunto de estados del AFND, buscando transiciones épsilon.
- Si un estado tiene una transición épsilon hacia otro estado, este último se une ("*U*") al primer estado y se busca si desde ese estado que se unió se pueden llegar a más estados.

Ejemplo:

[q0, "_", q1], [q0, "_", q2], [q1,"_", q3],[q2, a, q3]

Entonces las cláusulas épsilon serían:

[q0 U q1, q2, q3]

[q1 U q3]

[q2]

[q3]

2. Inicialización del Estado Inicial del AFD

El AFD comienza con un estado inicial que corresponde a la primera clausura épsilon. Esta clausura se convierte en el primer estado del AFD y se almacena en una lista (*estados_2*) que contiene los estados aún por procesar en el AFD.

3. Construcción Iterativa de Transiciones en el AFD

La conversión procede mediante una serie de pasos iterativos:

- **Recorrido de Estados Pendientes:** Se extrae el primer estado de la lista *estados_2* para procesarlo y construir sus transiciones.
- **Exploración de Símbolos del Alfabeto:** Para cada símbolo del alfabeto, se verifica si existe una transición en el AFND desde el estado actual.
- **Construcción de Estados Destino:** Si existe una transición, se construye un nuevo conjunto de estados destino. Para ello:
    - Se agrega el estado destino a una lista temporal (t).
    - Si el estado destino tiene una clausura epsilon, los estados alcanzables mediante esa clausura se agregan también a t.
- **Transición al Estado Sumidero:** Si no se encuentran transiciones válidas para un símbolo, el AFD transiciona al estado sumidero (Φ), aunque por el lazo agregado al principio en el AFND, esto nunca debería suceder y fue agregado para darle un uso universal al código.

4. Almacenamiento de Estados y Transiciones

- Cada transición construida se almacena en la lista *delta_min*, que representa las transiciones del AFD en el formato [*estado_origen*, símbolo, *estado_destino*].
- Si el estado destino no es el estado sumidero y aún no ha sido procesado ni está en la lista de estados pendientes, se añade a *estados_2* para ser procesado en la siguiente iteración.

5. Identificación de Estados Finales y Estados Totales en el AFD

- Los **estados finales** del AFD se definen como aquellos que contienen al menos un estado final del AFND.
- Los **estados totales** representan todos los estados alcanzables en el AFD, incluyendo el estado sumidero si se generó durante el proceso.

7. Ejemplos de uso:

Ejemplo 1: Expresión Regular Simple

*Ingrese su expresión regular: a**
Ingrese una Cadena: aaaa

Salida Esperada:

AFND
Estado Inicial q0
Estado final q3
Estados: ['q0', 'q1', 'q2', 'q3']
Transiciones: [['q0', '_', 'q1'], ['q1', 'a', 'q2'], ['q2', '_', 'q1'], ['q2', '_', 'q3'], ['q0', '_', 'q3']]
Diccionario: ['a']

Conversión del AFND a AFD:

Salida Esperada (AFD):

AFD
Clausulas Epsilon [['q0', 'U', 'q1', 'q3'], ['q1'], ['q2', 'U', 'q1', 'q3'], ['q3']]
Estado Inicial q0
Estados Finales [['q0', 'q1', 'q3'], ['q1', 'q2', 'q3', 'q0']]
Estados Totales [['q0', 'q1', 'q3'], ['q1', 'q2', 'q3', 'q0']]
Transiciones: [[['q0', 'q1', 'q3'], 'a', ['q1', 'q2', 'q3', 'q0']], [['q1', 'q2', 'q3', 'q0'], 'a', ['q1', 'q2', 'q3', 'q0']]]
Diccionario ['a']

Ejemplo 2: Expresión Regular Normal

*Ingrese su expresión regular: a|b**
Ingrese una Cadena: abaaabcbd

Salida Esperada:

AFND
Estado Inicial q0
Estado final q7
Estados: ['q0', 'q1', 'q2', 'q3', 'q4', 'q5', 'q6', 'q7']
Transiciones: [['q0', 'a', 'q0'], ['q0', 'b', 'q0'], ['q0', 'c', 'q0'], ['q0', 'd', 'q0'], ['q0', '_', 'q1'], ['q1', 'a', 'q2'], ['q2', '_', 'q7'], ['q0', '_', 'q3'], ['q3', '_', 'q4'], ['q4', 'b', 'q5'], ['q5', '_', 'q4'], ['q5', '_', 'q6'], ['q3', '_', 'q6'], ['q6', '_', 'q7']]
Diccionario ['a', 'b', 'c', 'd']

Conversión del AFND a AFD:

Salida Esperada (AFD):

AFD
Clausulas Epsilon [['q0', 'U', 'q1', 'q3'], ['q1'], ['q2', 'U', 'q7'], ['q3', 'U', 'q4', 'q6'], ['q4'], ['q5', 'U', 'q4', 'q6'], ['q6', 'U', 'q7'], ['q7']]
Estado Inicial q0
Estados Finales [['q1', 'q0', 'q3', 'q2', 'q7']]
Estados Totales [['q0', 'q1', 'q3'], ['q1', 'q0', 'q3', 'q2', 'q7'], ['q1', 'q3', 'q0']]
Transiciones: [[['q0', 'q1', 'q3'], 'a', ['q1', 'q0', 'q3', 'q2', 'q7']], [['q0', 'q1', 'q3'], 'b', ['q1', 'q3', 'q0']], [['q0', 'q1', 'q3'], 'c', ['q1', 'q3', 'q0']], [['q0', 'q1', 'q3'], 'd', ['q1', 'q3', 'q0']], [['q1', 'q0', 'q3', 'q2', 'q7'], 'a', ['q1', 'q0', 'q3', 'q2', 'q7']], [['q1', 'q0', 'q3', 'q2', 'q7'], 'b', ['q1', 'q3', 'q0']], [['q1', 'q0', 'q3', 'q2', 'q7'], 'c', ['q1', 'q3', 'q0']], [['q1', 'q0', 'q3', 'q2', 'q7'], 'd', ['q1', 'q3', 'q0']], [['q1', 'q3', 'q0'], 'a', ['q1', 'q0', 'q3', 'q2', 'q7']], [['q1', 'q3', 'q0'], 'b', ['q1', 'q3', 'q0']], [['q1', 'q3', 'q0'], 'c', ['q1', 'q3', 'q0']], [['q1', 'q3', 'q0'], 'd', ['q1', 'q3', 'q0']]]
Diccionario ['a', 'b', 'c', 'd']

8. Visualización

Para implementar una visualización en el código base que realiza la conversión de una Expresión Regular (ER) a un Autómata Finito No Determinista (AFND) y posteriormente a un Autómata Finito Determinista (AFD), fue necesario realizar diversas adaptaciones al código existente. Además, se incorporaron nuevas funcionalidades específicamente diseñadas para habilitar y gestionar el proceso de visualización de manera efectiva.

Código Actualizado del Programa Principal (ER_AFND_AFD.py):

```
1   class Thomson:
2       def __init__(self):
3           self.K = []  # Lista de estados
4           self.delta = []  # Lista de transiciones
5           self.lista_er = []  # Lista de la expresión regular
6           self.estado = 0  # Contador para los estados
7           self.primera_iteracion = True
8           self.cadena = []
9           self.diccionario = []
10
11      def juntar(self, er, agregar=""):
12          return er.replace("-", "") + agregar.replace("-", "")
13
14      def juntar_cadena(self, cadena, agregar=""):
15          return cadena.replace("-", "") + agregar.replace("-", "")
16
17      def crear_estado(self):
18          if self.primera_iteracion and self.estado != 0:
19              self.estado -= 1
20              estado = "q" + str(self.estado)
21              self.primera_iteracion = False
22              self.estado += 1
23              return estado
24          elif self.estado == 0:
25              estado = "q" + str(self.estado)
26              self.estado += 1
27              self.primera_iteracion = False
28              return estado
29          else:
30              estado = "q" + str(self.estado)
31              self.estado += 1
32              return estado
33
```

```
34      def agregar_estado(self, *estados):
35          for estado in estados:
36              if estado not in self.K:
37                  self.K.append(estado)
38
39      def simplificacion_er(self, er):
40          resultado = ""
41          i = 0
42          while i < len(er):
43              char = er[i]
44              if char not in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ*|.0Φ_":
45                  print(f"Carácter inválido '{char}' eliminado.")
46                  i += 1
47                  continue
48              if char in "|*." and resultado and resultado[-1] in "|*.":
49                  print(f"Operador consecutivo '{resultado[-1]}{char}' corregido a '{char}'.")
50                  resultado = resultado[:-1] + char
51              else:
52                  resultado += char
53              i += 1
54          if resultado.endswith("|") or resultado.endswith("."):
55              print(f"Operador '{resultado[-1]}' al final eliminado.")
56              resultado = resultado[:-1]
57          return resultado or "Φ"
58
59      def AFND(self, er):
60          if "-" in er:
61              er = self.juntar(er)
62          q_inicio = self.crear_estado()
63          for simbolo in sorted(self.diccionario):  # Orden alfabético del diccionario
64              self.delta.append([q_inicio, simbolo, q_inicio])
65          self.primera_iteracion = True
66          er = self.simplificacion_er(er)
67          self.lista_er = list(er)
68          i = 0
69          while i < len(self.lista_er):
70              if self.lista_er[i] not in ["Φ", "0"]:
71                  if i + 1 < len(self.lista_er) and self.lista_er[i + 1] == "*":
72                      q0, q1, q2, q3 = [self.crear_estado() for _ in range(4)]
73                      self.delta.extend([
74                          [q0, "_", q1],
75                          [q1, self.lista_er[i], q2],
76                          [q2, "_", q1],
77                          [q2, "_", q3],
```

```
78                  [q0, "_", q3]
79              ])
80              self.agregar_estado(q0, q1, q2, q3)
81              i += 2
82              self.primera_iteracion = True
83          elif self.lista_er[i] == "_":
84              q0 = self.crear_estado()
85              q1 = self.crear_estado()
86              self.delta.append([q0, "_", q1])
87              self.agregar_estado(q0, q1)
88              i += 1
89              self.primera_iteracion = True
90          elif i + 3 < len(self.lista_er) and self.lista_er[i + 1] == "|" and self.lista_er[i + 3] == "*":
91              q0, q1, q2, q3, q4, q5, q6, q7 = [self.crear_estado() for _ in range(8)]
92              self.delta.extend([
93                  [q0, "_", q1],
94                  [q1, self.lista_er[i], q2],
95                  [q2, "_", q7],
96                  [q0, "_", q3],
97                  [q3, "_", q4],
98                  [q4, self.lista_er[i + 2], q5],
99                  [q5, "_", q4],
100                 [q5, "_", q6],
101                 [q3, "_", q6],
102                 [q6, "_", q7]
103             ])
104             self.agregar_estado(q0, q1, q2, q3, q4, q5, q6, q7)
105             i += 4
106             self.primera_iteracion = True
107         elif i + 1 < len(self.lista_er) and self.lista_er[i + 1] == "|":
108             q0, q1, q2, q3, q4, q5 = [self.crear_estado() for _ in range(6)]
109             self.delta.extend([
110                 [q0, "_", q1],
111                 [q1, self.lista_er[i], q2],
112                 [q2, "_", q5],
113                 [q0, "_", q3],
114                 [q3, self.lista_er[i + 2], q4],
115                 [q4, "_", q5]
116             ])
117             self.agregar_estado(q0, q1, q2, q3, q4, q5)
118             if i + 3 < len(self.lista_er) and self.lista_er[i + 3] == "|":
119                 i += 2
120             else:
121                 i += 3
```

```
122                 self.primera_iteracion = True
123             else:
124                 q0 = self.crear_estado()
125                 q1 = self.crear_estado()
126                 self.delta.append([q0, self.lista_er[i], q1])
127                 self.agregar_estado(q0, q1)
128                 if i + 1 < len(self.lista_er) and self.lista_er[i + 1] == ".":
129                     i += 2
130                 else:
131                     i += 1
132                 self.primera_iteracion = True
133         else:
134             q_sumidero = self.crear_estado()
135             for simbolo in self.diccionario:
136                 self.delta.append([q_sumidero, simbolo, q_sumidero])
137             i = len(self.lista_er)
138 
139     def sigma(self, cadena):
140         # Limpiar cadena eliminando separadores innecesarios
141         cadena_limpia = cadena.replace("-", "").replace("\n", "")
142 
143         # Validar caracteres en el diccionario (permitir '_' como epsilon)
144         if any(char not in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_" for char in
145 cadena_limpia):
146             raise ValueError("El diccionario debe contener solo letras mayúsculas, minúsculas y '_'.")
147 
148         # Crear diccionario único y ordenado
149         self.diccionario = sorted(set(cadena_limpia))  # Incluye '_'
150 
151 
152 class Conversion:
153     def __init__(self, diccionario=None):
154         self.clausura_estado = []
155         self.delta_min = []
156         self.estados_finales = []
157         self.estados_totales = []
158         self.diccionario = sorted(diccionario) if diccionario else []
159         self.estado_inicial = "q0"  # Forzar el estado inicial a ser 'q0'
160 
161     def clausuras(self, estados, transiciones):
162         for m in range(len(estados)):
163             clausura_temporal = [estados[m]]
164             for transicion in transiciones:
165                 if estados[m] == transicion[0] and transicion[1] == "_":
```

```
166                 clausura_temporal.append(transicion[2])
167             # Remueve duplicados y ordena para consistencia
168             clausura_temporal = list(dict.fromkeys(clausura_temporal))
169             if len(clausura_temporal) > 1:
170                 clausura_temporal.insert(1, "U")  # Inserta 'U' solo si hay transiciones epsilon
171             self.clausura_estado.append(clausura_temporal)
172         self.AFD(transiciones, estados)
173 
174     def AFD(self, transiciones, estados):
175         estados_2 = [[self.estado_inicial]]
176         diccionario = sorted([t[1] for t in transiciones if t[1] != "_"])
177         procesados = set()
178         self.delta_min = []
179         estado_final_original = estados[-1]  # Estado final original del AFND
180 
181         while estados_2:
182             estado_actual = estados_2.pop(0)
183             estado_actual_tuple = tuple(sorted(estado_actual))
184             if estado_actual_tuple in procesados:
185                 continue
186             procesados.add(estado_actual_tuple)
187             for c in diccionario:
188                 t = set()
189                 for estado in estado_actual:
190                     for transicion in transiciones:
191                         if estado == transicion[0] and transicion[1] == c:
192                             destino = transicion[2]
193                             for clausura in self.clausura_estado:
194                                 if clausura[0] == destino:
195                                     t.update([s for s in clausura if s != "U"])
196                 t = list(t) or ["Φ"]
197                 t = sorted(t, key=lambda x: x if isinstance(x, str) else x[0])
198 
199                 # Verificar si hay transiciones válidas y si la transición no es redundante
200                 if len(t) > 1 or (len(t) == 1 and t[0] != "Φ"):
201                     sorted_estado_actual = sorted(estado_actual)
202                     if [sorted_estado_actual, c, t] not in self.delta_min:
203                         self.delta_min.append([sorted_estado_actual, c, t])
204                     if t not in estados_2 and t != ["Φ"]:
205                         estados_2.append(t)
206 
207             # Identificar estados finales
208             if any(estado_final_original in sub if isinstance(sub, list) else estado_final_original in estado_actual for
209 sub in estado_actual):
```

```
210             sorted_estado_actual = sorted(estado_actual)
211             if sorted_estado_actual not in self.estados_finales:
212                 self.estados_finales.append(sorted_estado_actual)
213 
214         # Corregir Estados Totales
215         self.estados_totales = []
216         for estado in procesados:
217             sorted_estado = sorted(list(estado))
218             # Filtrar redundantes y estados poco útiles
219             if len(sorted_estado) > 1 or sorted_estado == ['q0']:
220                 self.estados_totales.append(sorted_estado)
221 
222     def buscar_ocurrencias(self, texto):
223         ocurrencias = {}
224         lineas = texto.split("\n")
225 
226         for idx, linea in enumerate(lineas, start=1):
227             coincidencias = []
228             n = len(linea)
229             for inicio in range(n):
230                 estado_actual = self.estado_inicial
231                 subcadena = ""
232                 for fin in range(inicio, n):
233                     caracter = linea[fin]
234                     subcadena += caracter
235                     estado_siguiente = self._obtener_siguiente_estado(estado_actual, caracter)
236                     if estado_siguiente:
237                         estado_actual = estado_siguiente
238                         if estado_actual in self.estados_finales:
239                             # Registrar coincidencia con posición corregida
240                             coincidencias.append(f"{inicio + 2} {subcadena}")
241                     else:
242                         break  # Salir si no hay transición válida
243 
244             # Ordenar y eliminar duplicados
245             coincidencias = list(dict.fromkeys(coincidencias))
246             if coincidencias:
247                 ocurrencias[idx] = coincidencias
248 
249         return ocurrencias
250 
251     def _obtener_siguiente_estado(self, estado_actual, caracter):
252         for origen, simbolo, destino in self.delta_min:
253             if origen == estado_actual and simbolo == caracter:
```

```
254             return destino
255         return None
```

Figura 2: "Python, Código del programa de ejecución del proyecto Actualizado"

Código para generar la Interfaz (App.py):

```
1   import customtkinter as ctk
2   from tkinter import messagebox
3   from PIL import Image, ImageTk
4   from ER_AFND_AFD import Thomson, Conversion
5   from Visualizacion import Visualizador, Visualizador_2
6   from V_Busqueda import VisualizadorBusqueda
7   import os
8
9   # Configuración de apariencia
10  ctk.set_appearance_mode("light")
11  ctk.set_default_color_theme("dark-blue")
12
13  # Configuración de colores
14  FONDO_COLOR = "#656A6F"
15  COLOR_BOTON = "#8D8C8C"
16  COLOR_ENTRADA = "#dcdcdc"
17  COLOR_VISUAL = "#b0b0b0"
18
19  class ERApp(ctk.CTk):
20      def __init__(self):
21          super().__init__()
22          self.title("Programa Conversión ER")
23          self.geometry("1120x700")
24          self.configure(bg=FONDO_COLOR)
25
26          # Manejar eventos after
27          self.eventos_after = []
28
29          # Establecer el evento de cierre
30          self.protocol("WM_DELETE_WINDOW", self.on_closing)
31
32          # Establecer un icono personalizado
33          icon_path = os.path.join(os.path.dirname(__file__), "Imagenes/logo.ico")
34          self.iconbitmap(icon_path)
35
36          # Cargar imágenes
```

```
        base_path = os.path.dirname(__file__)
        ab_path = os.path.join(base_path, "Imagenes/a-b.png")
        abc_path = os.path.join(base_path, "Imagenes/abc.png")
        viz_path = os.path.join(base_path, "Imagenes/visualizacion.png")

        self.ab_img = ImageTk.PhotoImage(Image.open(ab_path).resize((60, 60), Image.Resampling.LANCZOS))
        self.abc_img = ImageTk.PhotoImage(Image.open(abc_path).resize((60, 60), Image.Resampling.LANCZOS))
        self.viz_img = ImageTk.PhotoImage(Image.open(viz_path).resize((400, 400), Image.Resampling.LANCZOS))

        # Instancias de conversión y visualización
        self.thomson = Thomson()
        self.converter = Conversion()
        self.visualizer = Visualizador(self)
        self.visualizer_2 = Visualizador_2(self)
        self.visualizador_busqueda = VisualizadorBusqueda(self)

        self.er = ctk.StringVar()
        self.text_input = ctk.StringVar()
        self.afnd_creado = False
        self.afd_creado = False
        self.er_ingresada = False
        self.cadena_ingresada = False

        self.crear_widgets()

    def crear_widgets(self):
        # Crear frames para organizar la disposición
        frame_superior = ctk.CTkFrame(self, fg_color=FONDO_COLOR)
        frame_superior.grid(row=0, column=0, columnspan=3, padx=22, pady=10, sticky="ew")

        frame_entrada = ctk.CTkFrame(self, fg_color=FONDO_COLOR)
        frame_entrada.grid(row=1, column=0, columnspan=2, padx=20, pady=10, sticky="nsew")

        frame_salida = ctk.CTkFrame(self, fg_color=FONDO_COLOR)
        frame_salida.grid(row=2, column=0, columnspan=2, padx=20, pady=10, sticky="nsew")

        frame_derecha = ctk.CTkFrame(self, fg_color=FONDO_COLOR, width=400, height=400)
        frame_derecha.grid(row=1, column=2, rowspan=2, padx=20, pady=10, sticky="nsew")


        self.grid_columnconfigure(0, weight=1)
        self.grid_columnconfigure(1, weight=1)
        self.grid_columnconfigure(2, weight=1)
        self.grid_rowconfigure(0, weight=1)
```

```
81          self.grid_rowconfigure(1, weight=1)
82          self.grid_rowconfigure(2, weight=1)
83
84          # Botones de navegación - Configuración
85          self.btn_intro_er = self.crear_boton(frame_superior, "Introducir ER", self.introducir_er)
86          self.btn_er_afnd = self.crear_boton(frame_superior, "ER -> AFND", self.convertir_er_a_afnd)
87          self.btn_afnd_afd = self.crear_boton(frame_superior, "AFND -> AFD", self.convertir_afnd_a_afd)
88          self.btn_buscar_texto = self.crear_boton(frame_superior, "Buscar ER en Texto", self.buscar_ocurrencias)
89          self.btn_visual_afnd = self.crear_boton(frame_superior, "Visualizar AFND", self.visualizar_afnd)
90          self.btn_visual_afd = self.crear_boton(frame_superior, "Visualizar AFD", self.visualizar_afd)
91          self.btn_visual_busqueda = self.crear_boton(frame_superior, "Visualizar Búsqueda", self.visualizar_busqueda)
92
93
94          # Entrada para ER con imagen al lado
95          self.er_image_label = ctk.CTkLabel(frame_entrada, image=self.ab_img, text="", bg_color=FONDO_COLOR)
96          self.er_image_label.grid(row=0, column=0, padx=10, pady=10, sticky="w")
97          self.er_entry = self.crear_entrada(frame_entrada, self.er, "Colocar la Expresión Regular")
98
99          # Entrada para buscar texto con imagen al lado
100         self.text_image_label = ctk.CTkLabel(frame_entrada, image=self.abc_img, text="", bg_color=FONDO_COLOR)
101         self.text_image_label.grid(row=1, column=0, padx=10, pady=10, sticky="w")
102         self.text_input_entry = ctk.CTkTextbox(frame_entrada, wrap="word", height=100, corner_radius=10,
103 fg_color=COLOR_ENTRADA)
104         self.text_input_entry.grid(row=1, column=1, columnspan=3, padx=10, pady=10, sticky="ew")
105
106         # Texto "Salidas de resultados"
107         self.label_salida = ctk.CTkLabel(frame_salida, text="Salidas de resultados", font=("Helvetica", 20, "bold"),
108 bg_color=FONDO_COLOR, text_color="white", anchor="center")
109         self.label_salida.grid(row=0, column=0, padx=10, pady=(10, 0), sticky="nsew")
110         self.text_output = ctk.CTkTextbox(frame_salida, height=300, width=400, corner_radius=10,
111 fg_color=COLOR_ENTRADA)
112         self.text_output.grid(row=1, column=0, padx=10, pady=(40, 20), sticky="nsew")
113
114         # Botón "Limpiar Resultados"
115         self.btn_limpiar = ctk.CTkButton(frame_salida, text="Limpiar Resultados", command=self.limpiar_resultados,
116 width=100, height=35, corner_radius=10, fg_color=COLOR_BOTON, hover_color="#a0a0a0")
117         self.btn_limpiar.grid(row=1, column=0, padx=10, pady=0, sticky="n")
118
119         # Título "Salida visual de Conversión" en el frame derecho
120         self.label_salida_visual = ctk.CTkLabel(frame_derecha, text="Salida visual de Conversión", font=("Helvetica", 20,
121 "bold"), bg_color=FONDO_COLOR, text_color="white", anchor="center")
122         self.label_salida_visual.grid(row=0, column=0, padx=(130, 0), pady=10, sticky="n")
123
124         # Área de visualización ajustada en el frame_derecha
```

```
125         self.visual_area = ctk.CTkFrame(frame_derecha, fg_color=FONDO_COLOR)
126         self.visual_area.grid(row=1, column=0, padx=10, pady=10, sticky="nsew")
127 
128         # Botón de proceso de simulación
129         self.btn_simulacion = ctk.CTkButton(frame_derecha, text=" ", width=150, height=40, corner_radius=10,
130 command=self.simular_proceso, fg_color=FONDO_COLOR, hover_color="#a0a0a0")
131         self.btn_simulacion.grid(row=3, column=0, padx=(100, 0), pady=20, sticky="n")
132 
133     def crear_boton(self, frame, texto, comando, ancho=120, altura=40):
134         boton = ctk.CTkButton(frame, text=texto, width=ancho, height=altura, corner_radius=10, fg_color=COLOR_BOTON,
135 hover_color="#a0a0a0", command=comando)
136         boton.grid(row=0, column=len(frame.winfo_children()), padx=15, pady=10)
137         return boton
138 
139     def crear_entrada(self, frame, variable, placeholder):
140         entrada = ctk.CTkEntry(frame, textvariable=variable, placeholder_text=placeholder, width=300, height=50,
141 corner_radius=10, fg_color=COLOR_ENTRADA)
142         entrada.grid(row=0, column=1, columnspan=3, padx=10, pady=10, sticky="ew")
143         return entrada
144 
145     def introducir_er(self):
146         er = self.er.get().strip()
147         if not er:
148             self.text_output.insert("end", "No hay introducido una expresión regular aún.\n\n")
149             return
150         if "ñ" in er or "Ñ" in er:
151             self.mostrar_error("Expresión regular inválida: contiene 'ñ' o 'Ñ'.")
152             return
153         # Validar expresiones regulares vacías o incorrectas
154         if er in ["0", "Φ"] or any(sub in er for sub in ["0.0", "Φ.Φ", "Φ.0", "0.Φ"]):
155             self.mostrar_error("Expresión regular vacía o cadena inválida.")
156             return
157         # Validar caracteres permitidos en la ER
158         if any(char not in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ|.*_0Φ \n" for char in er):
159             self.mostrar_error("Solo se permite letras mayúsculas o minúsculas y operadores válidos (*, |, ., _, 0, Φ).")
160             return
161 
162         # Reemplazar combinaciones válidas para procesarlas
163         er = er.replace("0|", "_|").replace("|0", "|_").replace("Φ|", "_|").replace("|Φ", "|_")
164         er = er.replace("0", "").replace("Φ", "")
165 
166         # Manejar 0* y Φ* como aceptación de toda la cadena
167         cadena = self.text_input_entry.get("1.0", "end").strip()
168         if "0*" in er or "Φ*" in er:
```

```
169             if not cadena:
170                 self.mostrar_error("Debe ingresar una cadena para continuar.")
171             else:
172                 self.text_output.insert("end", "Acepta toda la cadena.\n\n")
173             return
174 
175         # Validar cadena para contener únicamente letras y épsilon
176         if any(char not in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_" for char in
177 cadena.replace("-", "").replace("\n", "")):
178             self.mostrar_error("La cadena debe contener solo letras mayúsculas, minúsculas, '-' con enter y '_'.")
179             return
180 
181         self.er_ingresada = True
182         self.cadena_ingresada = True
183         self.text_output.insert("end", "Se introdujo la expresión regular correctamente.\n\n")
184         self.afnd_creado = False
185         self.afd_creado = False
186         self.thomson = Thomson()
187         self.thomson.sigma(cadena)  # Generar diccionario con letras y '_'
188 
189 
190     def convertir_er_a_afnd(self):
191         if not self.er_ingresada or not self.cadena_ingresada:
192             self.mostrar_error("Debe ingresar la expresión regular y la cadena antes de convertir.")
193             return
194 
195         if "0*" in self.er.get() or "Φ*" in self.er.get():
196             self.mostrar_error("No se puede generar el Automata Finito No Determinista (AFND) es una ER vacía.")
197             return
198 
199         self.afnd_creado = True
200         self.thomson = Thomson()  # Crear nueva instancia
201         self.thomson.sigma(self.text_input_entry.get("1.0", "end").strip())  # Actualizar diccionario
202         self.thomson.AFND(self.er.get())  # Generar AFND
203         self.mostrar_resultados_afnd()
204 
205 
206     def visualizar_afnd(self):
207         if not self.afnd_creado:
208             self.mostrar_error("Debe generar el AFND primero.")
209         else:
210             self.visualizer.mostrar_grafo(
211                 self.thomson.delta,
212                 "AFND",
```

```
213             estado_inicial=self.thomson.K[0],
214             estados_finales=[self.thomson.K[-1]]
215         )
216
217     def visualizar_afd(self):
218         if not self.afd_creado:
219             self.mostrar_error("Debe generar primero el AFD.")
220         else:
221             # Preparar los datos del AFD
222             transiciones_preparadas, estados_finales_preparados, estado_inicial_preparado =
223 self.visualizer_2.preparar_datos_afd(
224                 self.converter.delta_min,
225                 self.converter.estados_finales,
226                 self.converter.estado_inicial
227             )
228
229             # Mostrar el grafo del AFD usando Visualizador_2
230             self.visualizer_2.mostrar_grafo(
231                 transiciones_preparadas,
232                 "AFD",
233                 estado_inicial=estado_inicial_preparado,
234                 estados_finales=estados_finales_preparados
235             )
236
237     def convertir_afnd_a_afd(self):
238         if not self.afnd_creado:
239             self.mostrar_error("Debe generar primero el AFND.")
240             return
241
242         if "0*" in self.er.get() or "Φ*" in self.er.get():
243             self.mostrar_error("No se puede generar el Automata Finito Determinista (AFD) es una ER vacía.")
244             return
245
246         self.afd_creado = True
247         self.converter = Conversion(self.thomson.diccionario)  # Pasar el diccionario
248         self.converter.clausuras(self.thomson.K, self.thomson.delta)
249         self.mostrar_resultados_afd()
250
251     def mostrar_resultados_afnd(self):
252         self.text_output.insert("end", "————————————————————————— AFND
253 ————————————————————————\n")
254         self.text_output.insert("end", f"Estado Inicial: {self.thomson.K[0]}\n")
255         self.text_output.insert("end", f"Estado final: {self.thomson.K[-1]}\n")
256         self.text_output.insert("end", f"Estados: {self.thomson.K}\n")
```

```
257         self.text_output.insert("end", f"Transiciones: {self.thomson.delta}\n")
258         self.text_output.insert("end", f"Diccionario: {self.thomson.diccionario}\n")
259 
260     def mostrar_resultados_afd(self):
261         self.text_output.insert("end", "———————————————————————— AFD
262 ————————————————————————\n")
263         self.text_output.insert("end", f"Clausulas Epsilon: {self.converter.clausura_estado}\n")
264         self.text_output.insert("end", f"Estado Inicial: {self.converter.estado_inicial}\n")  # Mostrar 'q0'
265         self.text_output.insert("end", f"Estados Finales: {self.converter.estados_finales}\n")
266         self.text_output.insert("end", f"Estados Totales: {self.converter.estados_totales}\n")
267         self.text_output.insert("end", f"Transiciones: {self.converter.delta_min}\n")
268         self.text_output.insert("end", f"Diccionario: {self.converter.diccionario}\n")
269 
270     def buscar_ocurrencias(self):
271         er = self.er.get().strip()
272         texto = self.text_input_entry.get("1.0", "end").strip()
273 
274         if not er or not texto:
275             self.mostrar_error("Debe ingresar la ER y la cadena antes de buscar.")
276             return
277 
278         # Remover `_` del texto para la búsqueda
279         texto = texto.replace("_", "")
280 
281         # Validar caracteres permitidos en la cadena de búsqueda
282         if any(char not in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ\n-" for char in
283 texto.replace("-", "").replace("\n", "")):
284             self.mostrar_error("La cadena debe contener solo letras mayúsculas o minúsculas, '_', '-' con Enter")
285             return
286 
287         # Reemplazar en la ER los casos permitidos
288         er = er.replace("0|", "_|").replace("|0", "|_").replace("Φ|", "_|").replace("|Φ", "|_")
289         er = er.replace("0", "").replace("Φ", "")
290 
291         ocurrencias = self._encontrar_ocurrencias(er, texto)
292         self.text_output.insert("end", "———————————————————— Ocurrencias ————————————————————\n")
293         if ocurrencias:
294             for linea, posiciones in ocurrencias.items():
295                 posiciones_str = ', '.join([f'{pos[0]} "{pos[1]}"' for pos in posiciones])
296                 self.text_output.insert("end", f"Línea {linea}: {posiciones_str}\n")
297         else:
298             self.text_output.insert("end", "No se encontraron ocurrencias.\n")
299 
299 
```

```
300
301
302     def _encontrar_ocurrencias(self, er, texto):
303         ocurrencias = {}
304         lineas = texto.split("\n")
305
306         # Define una función para procesar patrones con operadores básicos
307         def match_pattern(pattern, cadena):
308             # Manejo de unión
309             if "|" in pattern:
310                 subpatterns = pattern.split("|")
311                 return any(match_pattern(sub, cadena) for sub in subpatterns)
312
313             # Manejo de concatenación explícita
314             if "." in pattern:
315                 partes = pattern.split(".")
316                 idx = 0
317                 for parte in partes:
318                     if idx < len(cadena) and cadena[idx:idx + len(parte)] == parte:
319                         idx += len(parte)
320                     else:
321                         return False
322                 return idx == len(cadena)
323
324             # Manejo de Kleene (*)
325             if len(pattern) > 1 and pattern[1] == "*":
326                 return match_pattern(pattern[2:], cadena) or (cadena and pattern[0] == cadena[0] and match_pattern(pattern,
327 cadena[1:]))
328
329             # Concatenación implícita
330             if not pattern:
331                 return not cadena
332
333             # Manejo de épsilon
334             if pattern[0] == "_":
335                 return match_pattern(pattern[1:], cadena)
336
337             # Coincidencia literal
338             return cadena and pattern[0] == cadena[0] and match_pattern(pattern[1:], cadena[1:])
339
340         # Procesa cada línea para encontrar coincidencias
341         for idx, linea in enumerate(lineas, start=1):
342             coincidencias = []
343             for inicio in range(len(linea)):
```

```
344             for fin in range(inicio + 1, len(linea) + 1):
345                 subcadena = linea[inicio:fin]
346                 if subcadena and match_pattern(er, subcadena):
347                     coincidencias.append((inicio + 1, subcadena))
348         if coincidencias:
349             ocurrencias[idx] = coincidencias
350 
351     return ocurrencias
352 
353 def mostrar_error(self, mensaje):
354     messagebox.showerror("Error", mensaje)
355 
356 def visualizar_busqueda(self):
357     if not self.afnd_creado or not self.afd_creado:
358         self.mostrar_error("Debe generar primero el AFND y el AFD antes de visualizar la búsqueda.")
359         return
360 
361     cadena = self.text_input_entry.get("1.0", "end").strip()
362     if not cadena:
363         self.mostrar_error("Debe ingresar una cadena para simular el proceso.")
364         return
365 
366     # Visualiza cómo el AFD procesa la cadena
367     self.visualizador_busqueda.mostrar_resultado_busqueda(
368         self.converter.delta_min,  # Transiciones del AFD
369         self.converter.estado_inicial,  # Estado inicial
370         self.converter.estados_finales,  # Estados finales
371         cadena,
372         "Simulación de Búsqueda en el AFD"
373     )
374 
375 
376 def limpiar_resultados(self):
377     self.text_output.delete("1.0", "end")
378     self.afnd_creado = False
379     self.afd_creado = False
380     self.er_ingresada = False
381     self.cadena_ingresada = False
382     self.thomson = Thomson()
383     self.converter = Conversion()
384     # Limpiar el área de visualización
385     for widget in self.visual_area.winfo_children():
386         widget.destroy()
387 
```

```
388     def simular_proceso(self):
389         self.text_output.insert("end", "Iniciando el proceso de simulación...\n")
390 
391     def programar_evento(self, delay, funcion):
392         evento_id = self.after(delay, funcion)
393         self.eventos_after.append(evento_id)
394 
395     def cancelar_eventos(self):
396         for evento_id in self.eventos_after:
397             try:
398                 self.after_cancel(evento_id)
399             except Exception:
400                 pass
401         self.eventos_after.clear()
402 
403     def on_closing(self):
404         self.cancelar_eventos()
405         self.destroy()
406 
407 if __name__ == "__main__":
408     app = ERApp()
409     app.mainloop()
```

Figura 3: "Python, Código de la aplicación App"

Código Visualización (Visualizacion.py):

```
1   import networkx as nx
2   import matplotlib.pyplot as plt
3   from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
4   from collections import defaultdict
5   import random
6
7   class Visualizador:
8       def __init__(self, parent):
9           self.parent = parent
10          self.tamano_nodos = 300
11          self.color_nodo_inicio = 'lightgreen'  # Color para el estado inicial
12          self.color_nodo_final = 'yellow'  # Color para los estados finales
13          self.color_nodo_intermedio = 'lightblue'  # Color para los nodos intermedios
14          self.curvatura_bucle = 0.3  # Curvatura para bucles
15          self.curvatura_arista = 0.3  # Curvatura para aristas entre nodos diferentes
16
17      def mostrar_grafo(self, transiciones, titulo, estado_inicial, estados_finales):
18          grafo = nx.DiGraph()
19          etiquetas_aristas = defaultdict(list)
20
21          # Agregar transiciones al grafo
22          for inicio, etiqueta, fin in transiciones:
23              etiqueta = 'ε' if etiqueta == '_' else etiqueta  # Usar 'ε' para transiciones epsilon
24              etiquetas_aristas[(inicio, fin)].append(etiqueta)
25
26          print("\n=== Depuración de transiciones ===")
27          for (inicio, fin), etiquetas in etiquetas_aristas.items():
28              etiqueta_combinada = ",".join(sorted(set(etiquetas)))
29              print(f"Inicio: {inicio}, Fin: {fin}, Etiqueta: {etiqueta_combinada}")
30              grafo.add_edge(inicio, fin, label=etiqueta_combinada)
31
32          # Configuración de colores para los nodos
33          colores_nodos = [
34              self.color_nodo_inicio if nodo == estado_inicial else
35              self.color_nodo_final if nodo in estados_finales else
36              self.color_nodo_intermedio
37              for nodo in grafo.nodes()
38          ]
39
40          # Crear la figura para el grafo
41          fig, ax = plt.subplots(figsize=(8, 5))
42          ax.set_title(titulo, fontsize=14)
```

```
43
44      # Posicionar los nodos
45      posicion = self._calcular_posicion(grafo, estado_inicial, estados_finales)
46
47      # Dibujar nodos
48      nx.draw(
49          grafo, posicion, with_labels=True,
50          node_size=self.tamano_nodos,
51          node_color=colores_nodos,
52          font_size=9,
53          font_weight='bold',
54          ax=ax
55      )
56
57      # Dibujar aristas
58      for (u, v), etiquetas in list(etiquetas_aristas.items()):
59          # Evitar dibujar la línea central en transiciones bidireccionales
60          if (v, u) in etiquetas_aristas:
61              etiqueta_u_v = ",".join(sorted(set(etiquetas_aristas.pop((u, v), []))))
62              etiqueta_v_u = ",".join(sorted(set(etiquetas_aristas.pop((v, u), []))))
63
64              # Dibujar flecha de u -> v
65              nx.draw_networkx_edges(
66                  grafo, posicion,
67                  edgelist=[(u, v)],
68                  connectionstyle=f'arc3,rad={self.curvatura_arista}',
69                  ax=ax
70              )
71              x1, y1 = posicion[u]
72              x2, y2 = posicion[v]
73              x_offset_uv = (x1 + x2) / 2
74              y_offset_uv = (y1 + y2) / 2 + 0.3  # Ajuste vertical para u -> v
75              ax.text(
76                  x_offset_uv, y_offset_uv, etiqueta_u_v,
77                  fontsize=7, ha='center', va='center',
78                  bbox=dict(facecolor='white', edgecolor='none', pad=0.2)
79              )
80
81              # Dibujar flecha de v -> u
82              nx.draw_networkx_edges(
83                  grafo, posicion,
84                  edgelist=[(v, u)],
85                  connectionstyle=f'arc3,rad={-self.curvatura_arista}',
86                  ax=ax
```

```
87              )
88              x_offset_vu = (x1 + x2) / 2
89              y_offset_vu = (y1 + y2) / 2 - 0.3  # Ajuste vertical para v -> u
90              ax.text(
91                  x_offset_vu, y_offset_vu, etiqueta_v_u,
92                  fontsize=7, ha='center', va='center',
93                  bbox=dict(facecolor='white', edgecolor='none', pad=0.1)
94              )
95
96          else:
97              # Dibujar flechas normales para transiciones no bidireccionales
98              estilo = self._determinar_estilo_arista(grafo, u, v)
99              etiqueta_combinada = ",".join(sorted(set(etiquetas)))  # Combinar etiquetas sin duplicados
100             nx.draw_networkx_edges(
101                 grafo, posicion,
102                 edgelist=[(u, v)],
103                 connectionstyle=estilo,
104                 ax=ax
105             )
106             x1, y1 = posicion[u]
107             x2, y2 = posicion[v]
108             x_offset = (x1 + x2) / 2
109             y_offset = (y1 + y2) / 2
110             ax.text(
111                 x_offset, y_offset, etiqueta_combinada,
112                 fontsize=7, ha='center', va='center',
113                 bbox=dict(facecolor='white', edgecolor='none', pad=0.2)
114             )
115
116     # Ajustar márgenes y mostrar
117     ax.margins(0)
118     plt.tight_layout()
119     # Limpiar área de visualización antes de dibujar
120     for widget in self.parent.visual_area.winfo_children():
121         widget.destroy()
122     canvas = FigureCanvasTkAgg(fig, self.parent.visual_area)
123     canvas.get_tk_widget().grid(row=1, column=0, padx=20, pady=10, sticky="nsew")
124     canvas.draw()
125
126 def _calcular_posicion(self, grafo, estado_inicial, estados_finales):
127     nodos = list(grafo.nodes())
128     posicion = {}
129     espacio_horizontal = 6  # Espacio entre niveles horizontales
130     espacio_vertical = 3  # Espacio entre nodos en el mismo nivel
```

```
131
132         # Posicionar estado inicial
133         posicion[estado_inicial] = (0, 0)
134
135         # Posicionar nodos intermedios
136         intermedios = [nodo for nodo in nodos if nodo not in [estado_inicial] + estados_finales]
137         finales = estados_finales
138
139         # Distribuir intermedios horizontalmente en el centro
140         num_intermedios = len(intermedios)
141         x_centro = len(nodos) * espacio_horizontal // 2  # Centrar horizontalmente
142         for idx, nodo in enumerate(intermedios):
143             x_offset = random.uniform(-13, 13)  # Pequeña aleatoriedad horizontal
144             y_offset = random.uniform(-idx, idx)  # Evitar superposición vertical
145             posicion[nodo] = (x_centro + x_offset, -idx * espacio_vertical + y_offset)
146
147         # Posicionar nodos finales
148         for idx, nodo in enumerate(finales):
149             posicion[nodo] = (len(nodos) * espacio_horizontal, 0)
150
151         return posicion
152
153     def _determinar_estilo_arista(self, grafo, nodo1, nodo2):
154         if nodo1 == nodo2:
155             return f'arc3,rad={self.curvatura_bucle}'
156         elif grafo.has_edge(nodo2, nodo1):
157             return f'arc3,rad={self.curvatura_arista}'
158         else:
159             return 'arc3,rad=0.0'
160
161     def set_tamano_nodos(self, tamano):
162         self.tamano_nodos = tamano
163
164     def set_color_nodo_inicio(self, color):
165         self.color_nodo_inicio = color
166
167     def set_color_nodo_final(self, color):
168         self.color_nodo_final = color
169
170     def set_color_nodo_intermedio(self, color):
171         self.color_nodo_intermedio = color
172
173     def set_curvatura_bucle(self, curvatura):
174         self.curvatura_bucle = curvatura
```

```
175
176     def set_curvatura_arista(self, curvatura):
177         self.curvatura_arista = curvatura
178
179
180 class Visualizador_2:
181     def __init__(self, parent):
182         self.parent = parent
183         self.tamano_nodos = 300
184         self.color_nodo_inicio = 'lightgreen'  # Color para el estado inicial
185         self.color_nodo_siguiente = 'limegreen'  # Color para el nodo siguiente al inicial
186         self.color_nodo_final = 'yellow'  # Color para los estados finales
187         self.color_nodo_intermedio = 'lightblue'  # Color para los nodos intermedios
188         self.curvatura_bucle = 0.3  # Curvatura para bucles
189         self.curvatura_arista = 0.3  # Curvatura para aristas entre nodos diferentes
190
191     def mostrar_grafo(self, transiciones, titulo, estado_inicial, estados_finales):
192         grafo = nx.DiGraph()
193         etiquetas_aristas = defaultdict(list)
194
195         # Agregar transiciones al grafo
196         for inicio, etiqueta, fin in transiciones:
197             etiqueta = 'ε' if etiqueta == '_' else etiqueta  # Usar 'ε' para transiciones epsilon
198             etiquetas_aristas[(tuple(inicio), tuple(fin))].append(etiqueta)
199
200         # Asegurar que todos los estados están en el grafo
201         for transicion in transiciones:
202             grafo.add_node(tuple(transicion[0]))
203             grafo.add_node(tuple(transicion[2]))
204
205         # Identificar el nodo siguiente al inicial
206         nodo_siguiente = None
207         for transicion in transiciones:
208             if tuple(transicion[0]) == tuple(estado_inicial):
209                 nodo_siguiente = tuple(transicion[2]) if isinstance(transicion[2], list) else transicion[2]
210                 break
211
212         # Configuración de colores para los nodos
213         colores_nodos = [
214             self.color_nodo_inicio if nodo == tuple(estado_inicial) else
215             self.color_nodo_siguiente if nodo == nodo_siguiente else
216             self.color_nodo_final if nodo in [tuple(final) for final in estados_finales] else
217             self.color_nodo_intermedio
218             for nodo in grafo.nodes()
```

```
219         ]
220
221         # Crear la figura para el grafo
222         fig, ax = plt.subplots(figsize=(8, 5))
223         ax.set_title(titulo, fontsize=14)
224
225         # Posicionar los nodos
226         posicion = self._calcular_posicion(grafo, tuple(estado_inicial), [tuple(final) for final in estados_finales])
227
228         # Dibujar nodos
229         nx.draw(
230             grafo, posicion, with_labels=True,
231             node_size=self.tamano_nodos,
232             node_color=colores_nodos,
233             font_size=8,
234             font_weight='bold',
235             ax=ax
236         )
237
238         # Dibujar aristas
239         for (u, v), etiquetas in list(etiquetas_aristas.items()):
240             estilo = self._determinar_estilo_arista(grafo, u, v)
241             etiqueta_combinada = ",".join(sorted(set(etiquetas)))  # Combinar etiquetas sin duplicados
242             nx.draw_networkx_edges(
243                 grafo, posicion,
244                 edgelist=[(u, v)],
245                 connectionstyle=estilo,
246                 ax=ax
247             )
248             x1, y1 = posicion[u]
249             x2, y2 = posicion[v]
250             x_offset = (x1 + x2) / 2
251             y_offset = (y1 + y2) / 2 + 0.5 if u == v else (y1 + y2) / 2  # Etiquetas de bucles más arriba
252             ax.text(
253                 x_offset, y_offset, etiqueta_combinada,
254                 fontsize=8, ha='center', va='center',
255                 bbox=dict(facecolor='white', edgecolor='none', pad=0.1)
256             )
257
258         # Ajustar márgenes y mostrar
259         ax.margins(0.1)
260         plt.tight_layout()
261
262         # Limpiar área de visualización antes de dibujar
```

```
263         for widget in self.parent.visual_area.winfo_children():
264             widget.destroy()
265         canvas = FigureCanvasTkAgg(fig, self.parent.visual_area)
266         canvas.get_tk_widget().grid(row=1, column=0, padx=20, pady=10, sticky="nsew")
267         canvas.draw()
268 
269     def _calcular_posicion(self, grafo, estado_inicial, estados_finales):
270         nodos = list(grafo.nodes())
271         posicion = {}
272         espacio_horizontal = 4  # Espacio horizontal entre nodos
273         espacio_vertical = 4  # Espacio vertical entre nodos
274 
275         # Posicionar estado inicial
276         posicion[estado_inicial] = (0, 0)
277 
278         # Posicionar nodos con aleatoriedad
279         intermedios = [nodo for nodo in nodos if nodo not in [estado_inicial] + estados_finales]
280         for nodo in intermedios:
281             while True:
282                 x_random = random.uniform(-len(nodos), len(nodos)) * espacio_horizontal
283                 y_random = random.uniform(-len(nodos), len(nodos)) * espacio_vertical
284                 overlap = any(
285                     (x_random, y_random) == posicion.get(n) for n in posicion
286                 )
287                 if not overlap:
288                     posicion[nodo] = (x_random, y_random)
289                     break
290 
291         # Posicionar nodos finales
292         for idx, nodo in enumerate(estados_finales):
293             posicion[nodo] = (len(nodos) * espacio_horizontal, idx * espacio_vertical)
294 
295         return posicion
296 
297     def _determinar_estilo_arista(self, grafo, nodo1, nodo2):
298         if nodo1 == nodo2:
299             return f'arc3,rad={self.curvatura_bucle}'
300         elif grafo.has_edge(nodo2, nodo1):
301             return f'arc3,rad={self.curvatura_arista}'
302         else:
303             return 'arc3,rad=0.1'
304 
305     def preparar_datos_afd(self, transiciones, estados_finales, estado_inicial):
306         transiciones_preparadas = []
```

```
307     for inicio, etiqueta, fin in transiciones:
308         inicio_hashable = tuple(inicio) if isinstance(inicio, list) else inicio
309         fin_hashable = tuple(fin) if isinstance(fin, list) else fin
310         transiciones_preparadas.append((inicio_hashable, etiqueta, fin_hashable))
311
312     estados_finales_preparados = [tuple(estado) if isinstance(estado, list) else estado for estado in estados_finales]
313     estado_inicial_preparado = tuple(estado_inicial) if isinstance(estado_inicial, list) else estado_inicial
314
315     return transiciones_preparadas, estados_finales_preparados, estado_inicial_preparado
```

Figura 4: "Python, Código de la aplicación 'Visualización' "

8.1 Funcionamiento de la Interfaz

La visualización de los autómatas finitos en este sistema depende de una serie de botones en la interfaz que guían al usuario a través del proceso. Estos botones permiten la generación y presentación gráfica de los autómatas, asegurando una experiencia clara y ordenada. A continuación, se describen los botones necesarios para la visualización y su papel en el flujo de trabajo (se debe destacar que el código que ya estaba hecho de *ER_AFND_AFD.py* se adaptó para que pudiera congeniar y funcionar de manera correcta y coherente en la interfaz):

Figura 5: "Interfaz, Aplicación de Conversión de Expresión Regular"

Introducir Expresión Regular (ER)
Este botón permite al usuario ingresar la expresión regular que define el lenguaje que el autómata debe procesar. Es el primer paso en el sistema, ya que sin esta entrada inicial, no se puede proceder con la construcción del autómata.

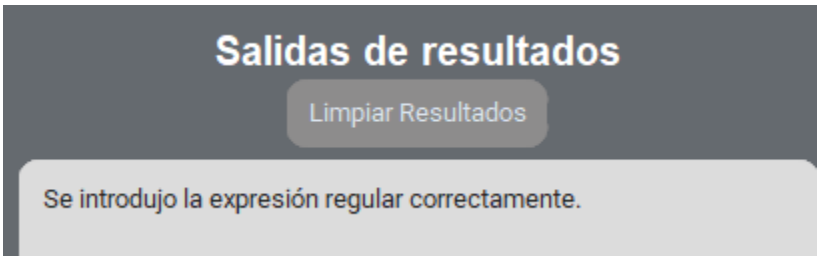

Figura 6: "Interfaz, Introducción de la Expresión Regular 'a|b' "

Convertir ER → AFND

Tras ingresar la expresión regular, este botón transforma dicha expresión en un autómata finito no determinista (AFND) utilizando el algoritmo de Thomson. Este autómata es una representación inicial que incluye transiciones epsilon y estados no deterministas.

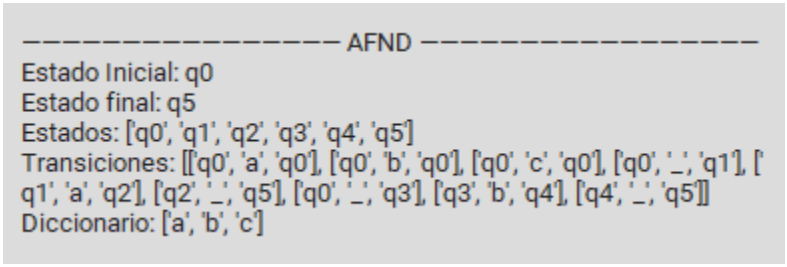

Figura 7: "Interfaz, Conversión de la ER 'a|b' en un AFD con la cadena a,b,c"

Botón de Visualizar AFND

Este botón genera una representación gráfica del AFND en el área de visualización. Los estados del autómata se muestran como nodos, mientras que las transiciones se presentan como aristas etiquetadas con los símbolos correspondientes. Esta visualización ayuda a comprender cómo la expresión regular se traduce en un modelo no determinista.

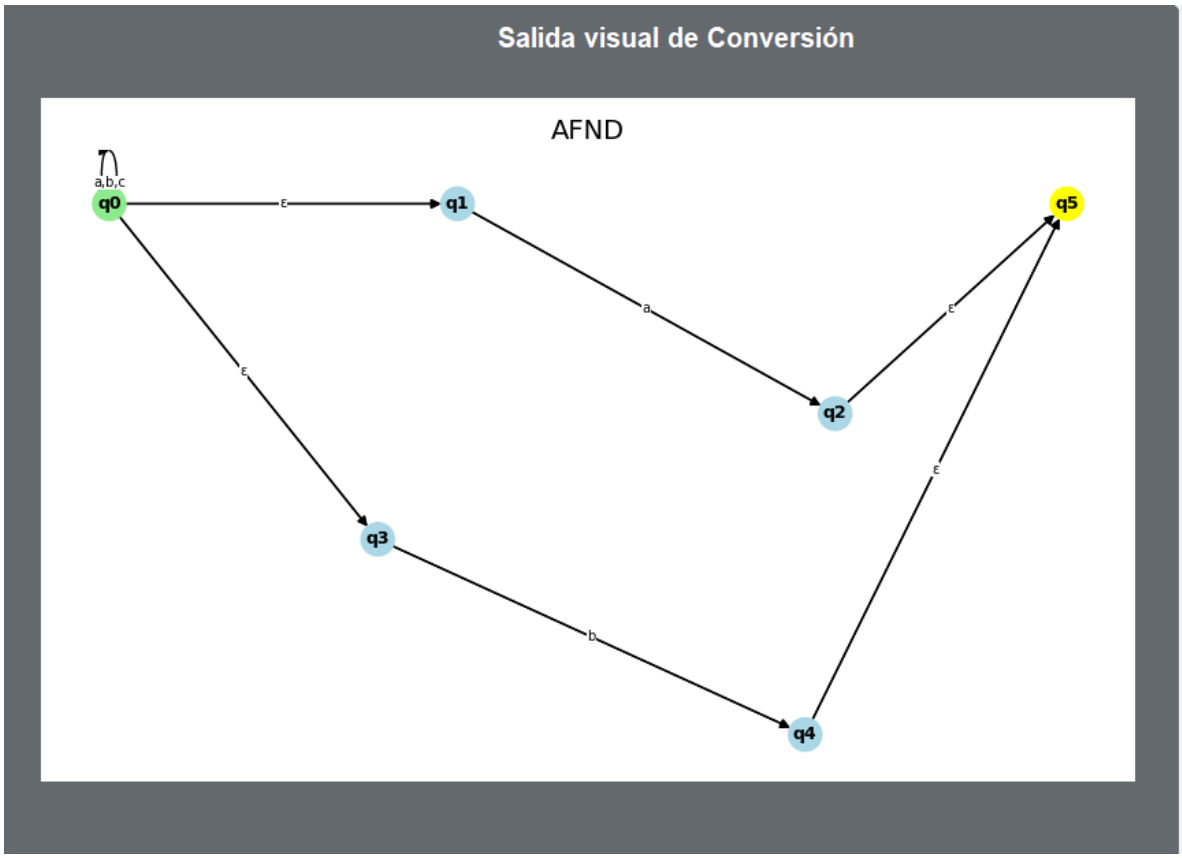

Figura 8: "Interfaz, Visualización del AFND 'a|b' "

El color verde identifica el estado inicial del autómata, mientras que los estados de transición se representan en azul. Por otro lado, el estado de aceptación se resalta en amarillo, indicando los estados finales donde las cadenas válidas son aceptadas.

También para comprobar si hay confusiones con la visualización en el terminal del compilador se muestra como son las ocurrencias de manera clara siendo para la visualización del Autómata Finito No Determinista lo siguiente:

```
1  === Depuración de transiciones ===
2  Inicio: q0, Fin: q0, Etiqueta: a,b,c
3  Inicio: q0, Fin: q1, Etiqueta: ε
4  Inicio: q1, Fin: q2, Etiqueta: a
5  Inicio: q2, Fin: q5, Etiqueta: ε
6  Inicio: q0, Fin: q3, Etiqueta: ε
7  Inicio: q3, Fin: q4, Etiqueta: b
```

8	Inicio: q4, Fin: q5, Etiqueta: ε

Figura 9: "Interfaz, Visualización del AFND 'a|b' "

Convertir AFND → AFD

Una vez generado el AFND, este botón lo convierte en un autómata finito determinista (AFD). El proceso elimina las transiciones epsilon y genera un autómata que responde de manera unívoca a cada símbolo de entrada, facilitando su análisis y uso en aplicaciones prácticas.

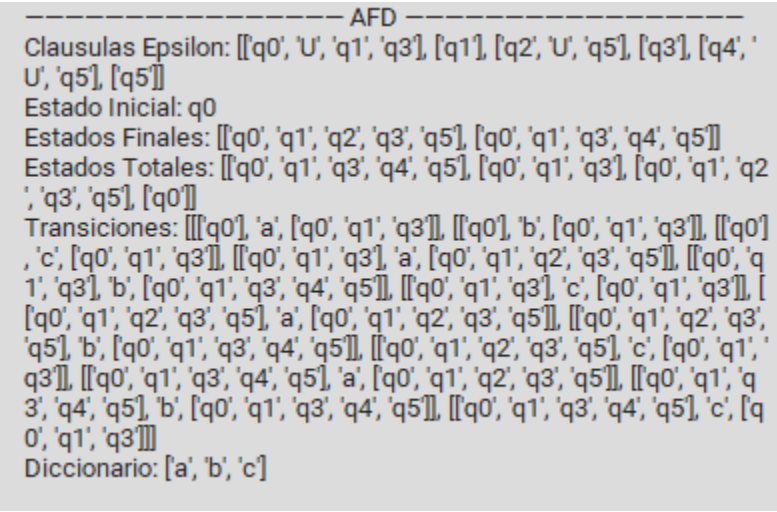

Figura 10: "Interfaz, Conversión del AFND en un AFD con la cadena a,b,c"

Visualizar AFD

Este botón muestra gráficamente el AFD resultante, organizando los estados y transiciones de manera clara. La visualización del AFD permite observar cómo se simplificó la estructura no determinista del AFND, obteniendo un modelo equivalente pero más eficiente.

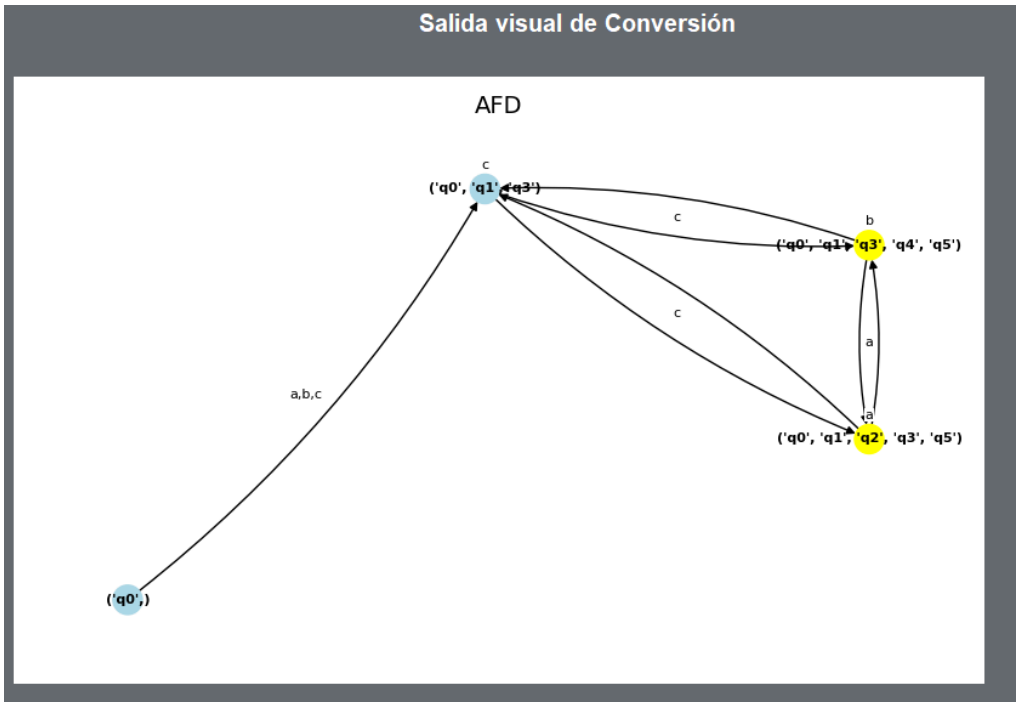

Figura 11: "Interfaz, Visualización del AFD 'a|b' "

Limpiar Resultados

Finalmente, este botón limpia los datos y la visualización actuales, permitiendo al usuario reiniciar el proceso desde cero. Aunque no afecta directamente la generación de autómatas, es esencial para mantener el orden y evitar confusiones al trabajar con nuevos datos.

8.2. Instrucciones de Compilación y Ejecución

Se detallan los requisitos, la instalación de dependencias y las instrucciones para ejecutar el programa. Es necesario instalar las siguientes extensiones de python en el cmd:

pip install graphviz
pip install pillow
pip install customtkinter
pip install networkx
pip install matplotlib

En linux se usa:
sudo apt-get install python3-tk

Es importante señalar que el funcionamiento debe ejecutarse paso a paso de manera correcta. Primeramente, se ejecuta la aplicación, la cual está llamada como "*App.py*" y se abrirá la interfaz. Luego se ingresa la expresión regular y su diccionario, al escribirlos luego se presiona arriba donde dice "*introducir ER*", luego se puede comenzar a trabajar pasándola a AFND y a AFD. Lo importante del programa es la visualización en grafos, ya que al pasar la ER a AFND se puede presionar la opción de arriba del menú llamado "*visualización*" (ya sea para AFND o AFD) y a continuación se mostrará un grafo en el espacio derecho de la interfaz con el autómata. Algo importante a destacar es que se puede presionar varias veces el botón de visualización y él autómata, cambiará de forma, se construirá de manera correcta, pero con diferentes combinaciones, haciendo que cada visualización sea diferente a la primera.

9. Ocurrencias

Las ocurrencias se refieren a todas las posiciones en un texto donde un patrón, descrito por una expresión regular, coincide. Este proceso consiste en buscar y detectar subcadenas en el texto de entrada que sean reconocidas por el autómata determinista generado (AFD). El objetivo de este módulo es demostrar la capacidad del programa para validar un patrón en cualquier posición del texto, proporcionando las coincidencias y las posiciones específicas.

Funcionamiento de las ocurrencias

Definición del patrón:

- El usuario ingresa una ER que describe el patrón a buscar.
- La ER se convierte en un AFND mediante el método de Thomson y luego en un AFD, que elimina la ambigüedad al garantizar un único camino para cada símbolo de entrada.

Procesamiento del texto:

- El texto de entrada se analiza línea por línea.
- Cada línea se recorre carácter por carácter, utilizando las transiciones del AFD.
- Cuando el AFD alcanza un estado final, significa que se ha encontrado una ocurrencia.

Resultados:

- Se registran las posiciones finales de cada coincidencia dentro del texto, indicando también el fragmento de texto que coincide con el patrón.

Implementación

El método "*buscar_ocurrencias*" implementado en la clase "*Conversion*" permite realizar este proceso. Se realiza de la siguiente manera:

Preparación del Texto: El texto se divide en líneas para procesar cada una por separado.

Iteración sobre Subcadenas: Por cada línea, se analiza cada subcadena posible, comenzando desde cada carácter inicial hasta el final de la línea.

Simulación del AFD: Usando el autómata finito determinista generado previamente, se evalúa cada carácter de la subcadena para verificar si lleva a un estado final.

Registro de Coincidencias: Si se alcanza un estado final al procesar una subcadena, se almacena la posición inicial y la subcadena coincidente.

El método devuelve un diccionario donde las claves son los números de línea y los valores son listas de subcadenas encontradas junto con sus posiciones.

El siguiente fragmento del código implementa esta funcionalidad:

```
1   def buscar_ocurrencias(self, texto):
2       ocurrencias = {}
3       lineas = texto.split("\n")
4
5       for idx, linea in enumerate(lineas, start=1):
6           coincidencias = []
7           n = len(linea)
8           for inicio in range(n):
9               estado_actual = self.estado_inicial
10              subcadena = ""
11              for fin in range(inicio, n):
12                  caracter = linea[fin]
13                  subcadena += caracter
14                  estado_siguiente = self._obtener_siguiente_estado(estado_actual, caracter)
15                  if estado_siguiente:
16                      estado_actual = estado_siguiente
17                      if estado_actual in self.estados_finales:
18                          # Registrar coincidencia con posición corregida
19                          coincidencias.append(f"{inicio + 2} {subcadena}")
20                  else:
21                      break  # Salir si no hay transición válida
22
23          # Ordenar y eliminar duplicados
24          coincidencias = list(dict.fromkeys(coincidencias))
25          if coincidencias:
26              ocurrencias[idx] = coincidencias
27
28      return ocurrencias
29
30  def _obtener_siguiente_estado(self, estado_actual, caracter):
31      for origen, simbolo, destino in self.delta_min:
32          if origen == estado_actual and simbolo == caracter:
33              return destino
34      return None
```

Figura 12: "Código 'Búsqueda de Ocurrencias'"

El programa utiliza el método *buscar_ocurrencias*, definido en la clase *Conversion*. Este método recorre el texto con el AFD generado, detectando las posiciones de las coincidencias y registrándolas en un formato estructurado.

Formato de salida: La salida de las ocurrencias incluye:

- El número de línea donde se encuentra la coincidencia.
- Las posiciones donde termina cada coincidencia.
- El texto correspondiente a la coincidencia de la Expresión regular.

Ejemplos de ejecución:

Ejemplo 1:

```
Expresión Regular: a*b
Cadena:
aaab-
ab-
b-
aaaa-
bbb-
ababab-

Ocurrencias encontradas:
Línea 1: 1 aaab
Línea 2: 1 ab
Línea 3: 1 b
Línea 5: 1 b
```

Ejemplo 2:

```
Expresión Regular: ab|ba

Texto:
abbaabba-
abab-
```

```
aaa-
babab-

Ocurrencias encontradas:
Línea 1: 1 ab, 4 ba
Línea 2: 1 ab
Línea 4: 2 ba
```

Las ocurrencias permiten aplicar las expresiones regulares para analizar grandes volúmenes de texto, detectando patrones definidos con precisión. Este sistema es crucial en aplicaciones como procesamiento de texto, búsqueda en documentos y análisis de datos.

10. Conclusión

Este proyecto logró implementar con éxito un sistema integral para convertir expresiones regulares en autómatas finitos, tanto no deterministas (AFND) como deterministas (AFD), utilizando los métodos y algoritmos definidos por Thomson y otros enfoques teóricos de la computación. La herramienta desarrollada no solo facilita la detección de patrones en texto, sino que también ofrece una interfaz gráfica intuitiva que permite visualizar y simular el funcionamiento de los autómatas generados, mejorando significativamente la comprensión del proceso.

El sistema demostró su utilidad al procesar diversas expresiones regulares de forma clara y precisa, encontrando coincidencias sin importar la complejidad del texto introducido. Esto permite convertir expresiones regulares en estructuras comprensibles y analíticas, representadas por autómatas que detectan patrones en texto de manera eficiente. A través de la transición de ER a AFND, se construyeron autómatas que soportan las operaciones básicas de concatenación, unión y estrella de Kleene. Posteriormente, mediante un algoritmo determinista, el AFND fue transformado en un AFD, eliminando la incertidumbre y garantizando un procesamiento más simple y efectivo.

Un aspecto destacado del proyecto es la integración de la interfaz gráfica, que no solo complementa la funcionalidad técnica del sistema, sino que también facilita al usuario interactuar con las expresiones regulares y observar los autómatas generados en tiempo real. Esta visualización interactiva, junto con la simulación de detección de patrones, proporciona una herramienta poderosa para quienes buscan entender o aplicar conceptos de teoría de autómatas en un entorno práctico y accesible.

En última instancia, el proyecto no solo cumplió con los requisitos establecidos, sino que demostró su practicidad y flexibilidad. Fue capaz de procesar eficientemente diversas expresiones regulares, encontrar coincidencias en textos complejos y ofrecer una experiencia visual enriquecedora. Esto consolida el sistema como una solución integral y eficaz para el procesamiento de texto basado en plantillas, validando tanto su utilidad como su potencial para aplicaciones futuras que requieran un sistema computacional como este.