Reporte de Práctica: Segundo Bloque de Programas de Teoría de la Computación

Serrano Gayosso José Eduardo
Grupo: 4CV2

30 de diciembre de 2024

Resumen

Este reporte presenta un análisis detallado de cuatro programas fundamentales diseñados para ilustrar conceptos avanzados de autómatas y lenguajes formales. Los programas incluyen un *Buscador de Palabras*, un *Autómata de Pila*, una implementación del esquema de *Backus-Naur* y una *Máquina de Turing*. Cada uno de estos programas aborda problemas específicos relacionados con la teoría de lenguajes, ofreciendo una perspectiva práctica que conecta conceptos teóricos con aplicaciones computacionales.

El desarrollo y ejecución de estos programas no solo ejemplifican el poder de los modelos computacionales, sino que también destacan la modularidad y flexibilidad de sus implementaciones. Para asegurar una visión integral, el reporte incluye el análisis detallado de sus componentes, la explicación de los métodos implementados, resultados obtenidos y su relevancia en el ámbito de la computación teórica.

Adicionalmente, se presenta un programa coordinador que permite ejecutar de manera controlada y eficiente los cuatro programas mencionados, reforzando su interacción e integración en un marco único.

Índice

1. Programa 1: Buscador de Palabras

1.1. Introducción

El programa `Buscador de Palabras` implementa un autómata finito determinista (AFD) para detectar y contar palabras específicas dentro de una cadena o archivo de texto. Este tipo de problema es común en el procesamiento de lenguaje natural, análisis de texto, y sistemas de búsqueda. El programa toma como entrada una tabla de transición de un autómata finito, procesa un conjunto de palabras predefinidas y genera un reporte con el conteo y las posiciones de las palabras encontradas.

Además, el programa incluye la capacidad de generar una representación gráfica del autómata, lo que facilita su análisis y comprensión.

1.2. AFND de Partida

El programa inicializa un *Autómata Finito No Determinista* (AFND) que sirve como base para procesar cadenas y detectar palabras clave. A continuación, se describe la estructura del AFND, su alfabeto, y las transiciones que definen su comportamiento.

1.2.1. Alfabeto

El alfabeto del autómata es el conjunto de caracteres ASCII. Esto incluye letras mayúsculas y minúsculas, números, y símbolos especiales.

1.2.2. Estados y Transiciones

El autómata tiene un conjunto de estados, incluyendo un estado inicial, estados intermedios, y estados de aceptación. La tabla de transiciones se define como sigue:

Cuadro 1: Tabla de Transiciones del AFND

Estado Actual	**Símbolo de Entrada**	**Estado Siguiente**
1	a	2, 7, 13
1	A	2, 7, 13
1	v	21, 28
1	V	21, 28
1	m	37
1	M	37
1	Σ	1

Estado Actual	Símbolo de Entrada	Estado Siguiente
2	*c*	3
3	*o*	4
4	*s*	5
5	*o*	**6**
7	*c*	8
8	*e*	9
9	*c*	10
10	*h*	11
11	*o*	**12**
13	*g*	14
14	*r*	15
15	*e*	16
16	*s*	17
17	*i*	18
18	*o*	19
18	ó	19
19	*n*	**20**
21	*i*	22
21	í	22
22	*c*	23
23	*t*	24
24	*i*	25
25	*m*	26
26	*a*	**27**
28	*i*	29
29	*o*	30
30	*l*	31
31	*a*	32
32	*c*	33
33	*i*	34
34	*o*	35
34	ó	35
35	*n*	**36**
37	*a*	38
38	*c*	39
39	*h*	40
40	*i*	41
41	*s*	42
42	*t*	43

Estado Actual	Símbolo de Entrada	Estado Siguiente
43	a	**44**

1.2.3. Representación Gráfica

Para complementar la tabla de transiciones, a continuación se muestra el diagrama de estados correspondiente:

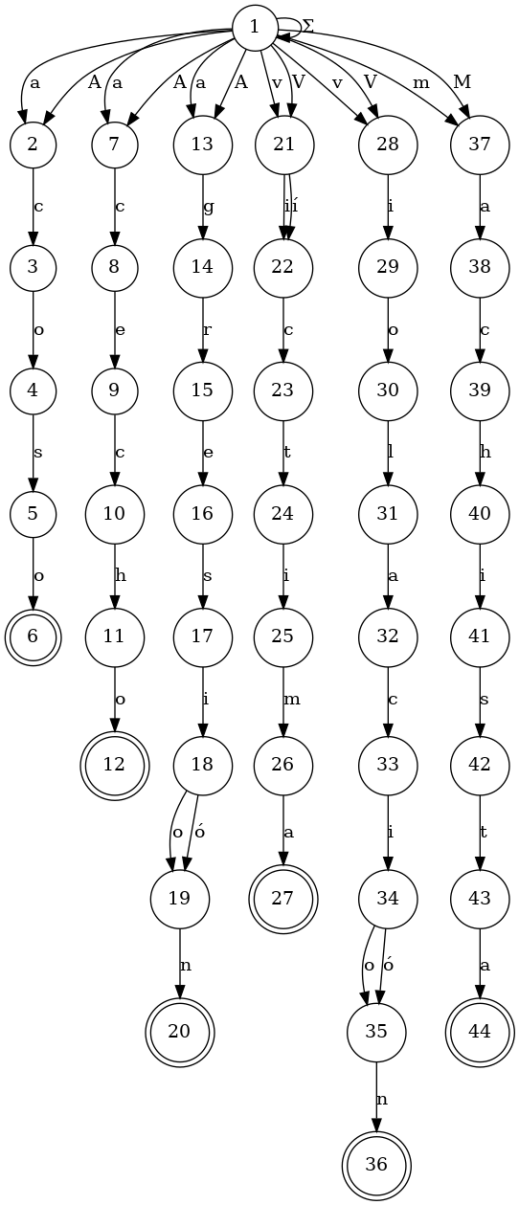

Figura 1: Autómata finito no determinista.

1.3. Tabla de Conjuntos de Estados del AFND

La siguiente tabla presenta la conversión del AFND al AFD, donde cada estado del AFD se representa como un subconjunto de estados del AFND. Este paso es esencial para comprender la construcción del AFD a partir del AFND.

estado/simbolo	a	c	e	g	h	i	l	m	n	o	r	s	t	v	ó	í	A	V	M
{1}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 2, 7, 13}	{1, 2, 7, 13}	{1, 3, 8}	{1}	{1, 14}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 37}	{1, 2, 7, 13, 38}	{1, 3, 8}	{1}	{1, 14}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 21, 28}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1, 22, 29}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1, 22}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 3, 8}	{1, 2, 7, 13}	{1}	{1, 9}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1, 4}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 14}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1, 15}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 2, 7, 13, 38}	{1, 2, 7, 13}	{1, 3, 8, 39}	{1}	{1, 14}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 22, 29}	{1, 2, 7, 13}	{1, 23}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1, 30}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 22}	{1, 2, 7, 13}	{1, 23}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 15}	{1, 2, 7, 13}	{1}	{1, 16}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 3, 8, 39}	{1, 2, 7, 13}	{1}	{1, 9}	{1}	{1, 40}	{1}	{1}	{1, 37}	{1}	{1, 4}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 30}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1, 31}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 23}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1, 24}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 16}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1, 17}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 40}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1, 41}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 9}	{1, 2, 7, 13}	{1, 10}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 4}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1, 5}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 31}	{1, 2, 7, 13, 32}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 24}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1, 25}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 17}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1, 18}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 41}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1, 42}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 10}	{1, 2, 7, 13}	{1}	{1}	{1}	{1, 11}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 5}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1, 6}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 2, 7, 13, 32}	{1, 2, 7, 13}	{1, 3, 8, 33}	{1}	{1, 14}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 25}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37, 26}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 18}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1, 19}	{1}	{1}	{1}	{1, 21, 28}	{1, 19}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 42}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1, 43}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 11}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1, 12}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 6}	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
{1, 3, 8, 33}	{1, 2, 7, 13}	{1}	{1, 9}	{1}	{1}	{1, 34}	{1}	{1, 37}	{1}	{1, 4}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 37, 26}	{1, 2, 7, 13, 38, 27}	{1, 3, 8}	{1}	{1, 14}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 19}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1, 20}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 43}	{1, 2, 7, 13, 44}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 12}	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
{1, 34}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1}	{1, 35}	{1}	{1}	{1}	{1, 21, 28}	{1, 35}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 2, 7, 13, 38, 27}	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
{1, 20}	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
{1, 2, 7, 13, 44}	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅
{1, 35}	{1, 2, 7, 13}	{1}	{1}	{1}	{1}	{1}	{1}	{1, 37}	{1, 36}	{1}	{1}	{1}	{1}	{1, 21, 28}	{1}	{1}	{1, 2, 7, 13}	{1, 21, 28}	{1, 37}
{1, 36}	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅

En esta tabla, cada estado AFD está representado por el conjunto de estados del AFND que lo componen. Este paso es crucial para construir la tabla final de transiciones del AFD, donde los estados se nombran de manera unitaria (e.g., $A, B, C, \ldots$).

1.4. Tabla de Transiciones del AFD

Se presenta ahora, la tabla final de transiciones del AFD con los nombres asignados a cada conjunto de estados para facilitar su lectura y la posterior construcción del diagrama.

estado / símbolo	**a**	**c**	**e**	**g**	**h**	**i**	**l**	**m**	**n**	**o**	**r**	**s**	**t**	**v**	**ó**	**í**	**A**	**V**	**M**
A	B	A	A	A	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
B	B	E	A	F	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
C	G	E	A	F	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
D	B	A	A	A	A	H	A	C	A	A	A	A	A	D	A	I	B	D	C
E	B	A	O	A	A	A	A	C	A	P	A	A	A	D	A	A	B	D	C
F	B	A	A	A	A	A	A	C	A	A	J	A	A	D	A	A	B	D	C
G	B	K	A	F	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
H	B	M	A	A	A	A	A	C	A	L	A	A	A	D	A	A	B	D	C
I	B	M	A	A	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
J	B	A	N	A	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
K	B	A	O	A	ñ	A	A	C	A	P	A	A	A	D	A	A	B	D	C
L	B	A	A	A	A	A	Q	C	A	A	A	A	A	D	A	A	B	D	C
M	B	A	A	A	A	A	A	C	A	A	A	A	R	D	A	A	B	D	C
N	B	A	A	A	A	A	A	C	A	A	A	S	A	D	A	A	B	D	C
ñ	B	A	A	A	A	T	A	C	A	A	A	A	A	D	A	A	B	D	C
O	B	U	A	A	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
P	B	A	A	A	A	A	A	C	A	A	A	V	A	D	A	A	B	D	C
Q	W	A	A	A	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
R	B	A	A	A	A	X	A	C	A	A	A	A	A	D	A	A	B	D	C
S	B	A	A	A	A	Y	A	C	A	A	A	A	A	D	A	A	B	D	C
T	B	A	A	A	A	A	A	C	A	A	A	Z	A	D	A	A	B	D	C
U	B	A	A	A	α	A	A	C	A	A	A	A	A	D	A	A	B	D	C
V	B	A	A	A	A	A	A	C	A	β	A	A	A	D	A	A	B	D	C
W	B	γ	A	F	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
X	B	A	A	A	A	A	A	ε	A	A	A	A	A	D	A	A	B	D	C
Y	B	A	A	A	A	A	A	C	A	ζ	A	A	A	D	ζ	A	B	D	C
Z	B	A	A	A	A	A	A	C	A	A	A	A	η	D	A	A	B	D	C
α	B	A	A	A	A	A	A	C	A	θ	A	A	A	D	A	A	B	D	C
β	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
γ	B	A	O	A	A	ι	A	C	A	P	A	A	A	D	A	A	B	D	C
ε	κ	E	A	F	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
ζ	B	A	A	A	A	A	A	C	λ	A	A	A	A	D	A	A	B	D	C
η	μ	A	A	A	A	A	A	C	A	A	A	A	A	D	A	A	B	D	C
θ	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
ι	B	A	A	A	A	A	A	C	A	ν	A	A	A	D	ν	A	B	D	C
κ	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
λ	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
μ	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
ν	B	A	A	A	A	A	A	C	ξ	A	A	A	A	D	A	A	B	D	C
ξ	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

1.5. Descripción del Programa

El programa está compuesto por diversos módulos que interactúan para cargar el autómata, procesar texto, generar reportes, y graficar el autómata. A continuación, se describe cada componente.

1.5.1. Métodos del Programa

A continuación, se describen de manera detallada los métodos principales del programa, enfatizando su funcionalidad y el flujo interno de cada uno.

`cargar_automata_desde_tabla(ruta_archivo)` Este método se encarga de cargar un autómata finito desde un archivo Excel que contiene la tabla de transición. A continuación, se describe su flujo interno:

1. **Lectura del archivo Excel:** Utilizando la biblioteca `pandas`, se carga la tabla de transición desde la ruta especificada. La primera columna de la tabla contiene los estados, y las demás columnas representan los símbolos del alfabeto.
2. **Inicialización del autómata:** Se crea un diccionario vacío llamado `automata`, donde cada clave será un estado, y el valor correspondiente será otro diccionario que asocia cada símbolo con el siguiente estado.
3. **Población del autómata:** Para cada fila de la tabla:
   - Se obtiene el estado actual de la primera columna.
   - Para cada símbolo del alfabeto (las demás columnas), se registra la transición al estado correspondiente en el diccionario.
4. **Retorno del autómata:** Al finalizar, el diccionario completamente construido se devuelve como resultado del método.

`procesar_cadena(automata, cadena, estado_inicial, estados_finales, palabras)` Este método procesa una cadena de entrada utilizando las reglas del autómata y devuelve un conteo de las palabras encontradas. Su flujo es el siguiente:

1. **Inicialización:** Se crea un contador para registrar la cantidad de veces que aparece cada palabra, inicializándolo en cero para todas las palabras en `palabras`.
2. **Inicio del procesamiento:** Se define el estado inicial como el estado actual y se comienza a recorrer la cadena carácter por carácter.

3. **Transiciones:**
    - Para cada símbolo, se verifica si existe una transición válida desde el estado actual.
    - Si no hay transición válida, el estado actual se reinicia al estado inicial y se continúa con el siguiente símbolo.
    - Si hay una transición válida, se actualiza el estado actual según las reglas del autómata.
4. **Detección de palabras:** Si se llega a un estado final:
    - Se incrementa el contador de la palabra asociada a ese estado final.
    - Dependiendo del estado final, el autómata puede reiniciarse al estado inicial o continuar desde un estado intermedio.
5. **Retorno del resultado:** Al finalizar, se devuelve el diccionario con el conteo de palabras.

`procesar_archivo(automata, ruta_texto, estado_inicial, estados_finales, palabras, archivo_historia)` Este método analiza un archivo de texto línea por línea, detecta las palabras definidas en el autómata, y registra un historial detallado del procesamiento. Su funcionamiento es el siguiente:

1. **Inicialización:**
    - Se inicializa un contador para cada palabra, similar al método anterior.
    - Se crean listas para registrar las posiciones (línea y columna) donde se encuentran las palabras.
    - Se abre un archivo de historial (`archivo_historia`) para registrar cada transición del autómata.
2. **Lectura del archivo:** Se abre el archivo de texto y se recorren sus líneas una por una.
3. **Procesamiento de cada línea:** Para cada carácter de una línea:
    - Se registra el estado previo y el estado siguiente en el archivo de historial.
    - Si se encuentra una palabra (estado final), se actualiza el contador correspondiente y se registra la posición (línea, columna) en la lista asociada a esa palabra.

- Si no hay transición válida, el autómata reinicia su análisis desde el estado inicial.

4. **Retorno de resultados:** Al finalizar, se devuelve el diccionario de conteos y las posiciones de las palabras encontradas.

`graficar_automata(automata, estado_inicial, estados_finales, nombre_archivo)`
Este método genera una representación gráfica del autómata en formato `png`.
Su flujo interno es el siguiente:

1. **Inicialización:** Se crea un objeto `graphviz.Digraph` para construir el grafo.

2. **Definición de los estados:**

   - Se recorren todos los estados del autómata.
   - Los estados finales se representan con doble círculo, y el estado inicial se marca con un indicador especial.

3. **Adición de transiciones:** Para cada estado y símbolo del alfabeto, se añade una transición al grafo, etiquetando las flechas con los símbolos correspondientes.

4. **Generación del archivo:** El grafo se renderiza y guarda en formato `png`.

`main()` El método principal coordina la ejecución del programa. Su flujo es el siguiente:

1. **Carga del autómata:** Se llama al método `cargar_automata_desde_tabla` para cargar la tabla de transición desde un archivo de Excel.

2. **Procesamiento de cadenas:** Se utiliza el método `procesar_cadena` para analizar una cadena definida en el código.

3. **Procesamiento de archivos:** Se llama al método `procesar_archivo` para analizar un archivo de texto. Los resultados se imprimen en la consola.

4. **Graficación del autómata:** Se genera una representación gráfica del autómata y se guarda en formato `png`.

5. **Resultados:** Se imprimen en la consola los conteos y posiciones de las palabras encontradas.

1.5.2. Ejecución Principal

El programa principal (`main`) coordina todas las operaciones. Permite:

- Procesar una cadena ingresada manualmente.
- Analizar un archivo de texto.
- Generar la representación gráfica del autómata.

1.6. Funcionamiento del Programa

El funcionamiento del programa se basa en el modelo de un AFD. A continuación, se describe el flujo detallado del procesamiento:

1. **Carga del Autómata:**
   - Se carga el autómata desde un archivo de Excel (`TablaAutomata.xlsx`) mediante el método `cargar_automata_desde_tabla`.
   - El autómata se representa como un diccionario donde cada estado contiene un conjunto de transiciones asociadas a los símbolos del alfabeto.
2. **Procesamiento de Cadenas:**
   - El programa recorre cada símbolo de la cadena, utilizando las reglas de transición del autómata para determinar el siguiente estado.
   - Si se llega a un estado final, se incrementa el contador asociado a la palabra correspondiente. Dependiendo del estado final alcanzado, el autómata puede reiniciar su análisis desde el estado inicial o continuar desde un estado intermedio.
3. **Procesamiento de Archivos:**
   - El programa analiza cada línea y carácter del archivo.
   - Registra el estado previo, el estado siguiente y la posición del carácter procesado en un archivo de historial.
   - Cuenta las palabras encontradas y almacena sus posiciones (línea y columna).
4. **Graficación del Autómata:**

- Se genera una imagen en formato png que ilustra la estructura del autómata.
- Cada estado y transición se etiqueta con los símbolos y estados correspondientes, destacando los estados iniciales y finales.

1.7. Código Fuente

```
import pandas as pd
import graphviz

def cargar_automata_desde_tabla(ruta_archivo):
    df = pd.read_excel(ruta_archivo)

    estados = df['estado\\simbolo']   # Primera columna tiene los estados
    simbolos = df.columns[1:]         # Todas las dem s columnas son s mbolos del alfabeto

    automata = {}
    for idx, estado in enumerate(estados):
        automata[estado] = {}
        for simbolo in simbolos:
            automata[estado][simbolo] = df.at[idx, simbolo]

    return automata

def procesar_cadena(automata, cadena, estado_inicial, estados_finales, palabras):
    contador = {palabra: 0 for palabra in palabras.values()}
    # diccionario
    estado_actual = estado_inicial

    for simbolo in cadena:
        if simbolo not in automata[estado_actual]:
            estado_actual = estado_inicial
            continue

        estado_actual = automata[estado_actual][simbolo]

        if estado_actual in estados_finales:
            palabra_encontrada = palabras[estado_actual]
            contador[palabra_encontrada] += 1
            if estado_actual == "  " or estado_actual == "  ":
                estado_actual = "G"
            else:
                estado_actual = estado_inicial
```

```
36
37      return contador
38
39
40  def procesar_archivo(automata, ruta_texto, estado_inicial,
    estados_finales, palabras, archivo_historia="historia.txt"
    ):
41      contador = {palabra: 0 for palabra in palabras.values()}
42      posiciones = {palabra: [] for palabra in palabras.values
    ()}
43
44      with open(archivo_historia, "w", encoding="utf-8") as
    hist:
45          hist.write("Historia de Procesamiento del Aut mata:\
    n")
46          hist.write("Car cter\tEstadoAnterior\
    tEstadoSiguiente\tPosici n(x,y)\n")
47
48          with open(ruta_texto, "r", encoding="utf-8") as f:
49              lineas = f.readlines()
50
51          estado_actual = estado_inicial
52          for y, linea in enumerate(lineas, start=1):
53              for x, simbolo in enumerate(linea, start=1):
54                  estado_anterior = estado_actual
55
56                  if simbolo not in automata[estado_actual]:
57                      estado_actual = estado_inicial
58                      hist.write(f"{repr(simbolo)}\t{
    estado_anterior}\t{estado_actual}\t({x},{y})\n")
59                      continue
60
61                  estado_siguiente = automata[estado_actual][
    simbolo]
62                  estado_actual = estado_siguiente
63
64                  hist.write(f"{repr(simbolo)}\t{
    estado_anterior}\t{estado_siguiente}\t({x},{y})\n")
65
66                  if estado_actual in estados_finales:
67                      palabra_encontrada = palabras[
    estado_actual]
68                      contador[palabra_encontrada] += 1
69                      posiciones[palabra_encontrada].append((x,
     y))
70
71                      if estado_actual == "   " or estado_actual
     == "   ":
72                          estado_actual = "G"
```

```
                    else:
                        estado_actual = estado_inicial

    return contador, posiciones


def graficar_automata(automata, estado_inicial, estados_finales, nombre_archivo="dfa"):

    dot = graphviz.Digraph(comment="DFA")
    dot.attr(rankdir="LR")  # Orientaci n de izquierda a derecha

    # Conjunto de todos los estados
    todos_estados = set(automata.keys())

    # A adir nodos para cada estado
    for estado in todos_estados:
        # Estado final con doble c rculo
        if estado in estados_finales:
            dot.node(estado, shape="doublecircle")
        # Estado inicial, en color azul o con alg n indicador
        elif estado == estado_inicial:
            dot.node(estado, shape="circle", color="blue")
        else:
            dot.node(estado, shape="circle")

    # Flecha invisible para marcar el estado inicial
    dot.node("ini", shape="point")
    dot.edge("ini", estado_inicial, label="inicio")

    # A adir transiciones
    for estado in automata:
        for simbolo, siguiente in automata[estado].items():
            dot.edge(estado, siguiente, label=simbolo)

    # Guardar el diagrama en PNG
    dot.render(filename=nombre_archivo, format="png", cleanup=True)
    print(f"Automata guardado en: {nombre_archivo}.png")

def main():
    ruta_archivo = "TablaAutomata.xlsx"
    automata = cargar_automata_desde_tabla(ruta_archivo)

    estado_inicial = "A"
    estados_finales = ["  ", "  ", "  ", "  ", "  ", "  "]
    palabras = {
```

```
        "   ": "Acoso",
        "   ": "Acecho",
        "   ": "Agresi n",
        "   ": "V ctima",
        "   ": "Violaci n",
        "   ": "Machista"
    }

    cadena = "victima, victimacecho V ctimagresion
    Victimacosov ctimachistacoso"
    resultado = procesar_cadena(automata, cadena,
    estado_inicial, estados_finales, palabras)

    print("Conteo de palabras encontradas (modo cadena
    directa):")
    for palabra, conteo in resultado.items():
        print(f"{palabra}: {conteo}")

    archivo_entrada = "texto_entrada.txt"
    archivo_salida_historia = "historia.txt"

    print("\nProcesando el archivo de texto...")

    conteo_archivo, posiciones_archivo = procesar_archivo(
        automata,
        archivo_entrada,
        estado_inicial,
        estados_finales,
        palabras,
        archivo_historia=archivo_salida_historia
    )

    print("Conteo de palabras encontradas en el archivo:")
    for palabra, cant in conteo_archivo.items():
        print(f"{palabra}: {cant}")

    print("\nPosiciones donde se encontraron las palabras en
    el archivo:")
    for palabra, lista_posiciones in posiciones_archivo.items
    ():
        if lista_posiciones:
            print(f"Palabra: {palabra}")
            for (x, y) in lista_posiciones:
                print(f"  - Encontrada en (x={x}, y={y})")


    print("\nGenerando imagen del aut mata...")
    graficar_automata(automata, estado_inicial,
    estados_finales, nombre_archivo="dfa")
```

```
161
162 if __name__ == "__main__":
163     main()
```

Listing 1: Código Fuente del Programa 1

1.8. Ejecución

A continuación, se describe el proceso de ejecución y los resultados observados.

1.8.1. Procesamiento de Cadenas

En este modo, se analiza una cadena ingresada por el usuario. A continuación, se muestra un ejemplo de salida:

```
Conteo de palabras encontradas (modo cadena directa):
Acoso: 2
Acecho: 1
Agresión: 1
Víctima: 2
Violación: 0
Machista: 1
```

1.8.2. Procesamiento de Archivos

El análisis de un archivo genera un reporte que incluye el conteo de palabras y sus posiciones. A continuación, se muestra un ejemplo:

```
Conteo de palabras encontradas en el archivo:
Acoso: 5
Acecho: 3
Agresión: 2
Víctima: 4
Violación: 1
Machista: 3

Posiciones donde se encontraron las palabras en el archivo:
Palabra: Acoso
  - Encontrada en (x=10, y=1)
  - Encontrada en (x=15, y=2)
...
```

1.8.3. Representación Gráfica del Autómata

La representación gráfica del autómata finito determinista es demasiado grande para entrar en este reporte, por lo que se encuentra en el siguiente hipervínculo: Imagen del AFD.

1.8.4. Historia del proceso

Por último, se muestra la evaluación del autómata por cada carácter que leyó y el cambio de estado.

```
Historia de Procesamiento del Autómata:
Carácter        EstadoAnterior  EstadoSiguiente Posición(x,y)
'L'     A       A       (1,1)
'a'     A       B       (2,1)
' '     B       A       (3,1)
'v'     A       D       (4,1)
'i'     D       H       (5,1)
'o'     H       L       (6,1)
'l'     L       Q       (7,1)
'e'     Q       A       (8,1)
'n'     A       A       (9,1)
'c'     A       A       (10,1)
'i'     A       A       (11,1)
'a'     A       B       (12,1)
' '     B       A       (13,1)
'c'     A       A       (14,1)
'o'     A       A       (15,1)
'n'     A       A       (16,1)
't'     A       A       (17,1)
'r'     A       A       (18,1)
'a'     A       B       (19,1)
' '     B       A       (20,1)
'l'     A       A       (21,1)
'a'     A       B       (22,1)
's'     B       A       (23,1)
' '     A       A       (24,1)
'm'     A       C       (25,1)
'u'     C       A       (26,1)
'j'     A       A       (27,1)
'e'     A       A       (28,1)
'r'     A       A       (29,1)
'e'     A       A       (30,1)
's'     A       A       (31,1)
' '     A       A       (32,1)
'e'     A       A       (33,1)
's'     A       A       (34,1)
' '     A       A       (35,1)
'u'     A       A       (36,1)
'n'     A       A       (37,1)
```

Figura 2: Archivo de texto que contiene el historial de evaluaciones.

2. Programa 2: Autómata de Pila

2.1. Introducción

El programa `Autómata de Pila` implementa un modelo de autómata de pila para analizar cadenas que pertenecen a un lenguaje definido por una gramática libre de contexto. En este caso, el lenguaje aceptado es $\{0^n1^n \mid n \geq 1\}$, que representa cadenas donde el número de ceros es igual al número de unos, y todos los ceros preceden a los unos.

Los autómatas de pila extienden las capacidades de los autómatas finitos al introducir una estructura adicional: la pila. Esta pila permite almacenar y recuperar información de manera dinámica, lo que los hace ideales para reconocer lenguajes libres de contexto. Su relevancia se observa en aplicaciones como análisis sintáctico en compiladores, procesamiento de texto, y validación de patrones complejos.

El programa utiliza una animación gráfica para ilustrar las transiciones del autómata, proporcionando una representación visual de la operación de la pila y el estado actual del autómata.

2.2. Descripción del Programa

El programa está compuesto por múltiples componentes que interactúan para validar una cadena. A continuación, se describen los elementos principales:

2.2.1. Clases Principales

- `Nodo:` Representa cada elemento de la pila, que contiene un dato y un puntero al siguiente nodo.
- `Pila:` Define las operaciones de la pila descritas anteriormente.
- `AutomataAnimado:` Gestiona la animación gráfica del autómata y visualiza las transiciones entre estados.

2.2.2. Estructura de Datos: Pila

La pila es una estructura fundamental en este programa. Implementada como una lista enlazada, admite operaciones básicas como:

- **push:** Inserta un elemento en el tope de la pila.
- **pop:** Elimina y devuelve el elemento en el tope de la pila.

- **peek:** Devuelve el elemento en el tope sin eliminarlo.
- **mostrarPila:** Devuelve el contenido actual de la pila en forma de lista.

2.2.3. `AutomataAnimado`

La clase `AutomataAnimado` se encarga de representar gráficamente las transiciones del autómata de pila, mostrando la evolución de los estados, la cadena restante y el contenido de la pila durante el proceso de aceptación o rechazo de la cadena. Este componente combina conceptos teóricos de los autómatas con herramientas visuales interactivas, facilitando la comprensión del funcionamiento del modelo.

Diseño de la clase La clase utiliza `Tkinter`, una biblioteca estándar de Python para crear interfaces gráficas. A continuación, se detallan los elementos principales:

- **Ventana principal:** Se inicializa con dimensiones predefinidas y un lienzo (`Canvas`) para dibujar los elementos visuales.
- **Transiciones:** Un atributo de la clase que contiene la secuencia completa de estados, cadena restante y contenido de la pila en cada paso del autómata.
- **Método `dibujarTransicion`:** Actualiza el lienzo en cada paso para reflejar la configuración actual del autómata.

Atributos de la clase La clase contiene los siguientes atributos clave:

- `self.cadena`: Almacena la cadena ingresada para procesar.
- `self.transiciones`: Una lista de tuplas (*estado*, cadena_restante, pila) que representa cada paso del autómata.
- `self.paso`: Un contador que rastrea el paso actual del proceso de animación.
- `self.canvas`: Un objeto gráfico donde se dibujan los estados, la pila y la cadena restante.

Método `dibujarTransicion` Este método es el núcleo de la animación. Se ejecuta en intervalos regulares para representar cada transición de forma visual. A continuación, se desglosan sus pasos:

1. **Borrado del lienzo:** En cada llamada al método, se elimina el contenido previo utilizando `self.canvas.delete(.ªll")`, asegurando que el lienzo esté listo para dibujar la siguiente transición.

2. **Dibujar el estado actual:** Se crea un rectángulo que representa el estado actual del autómata, acompañado de una etiqueta con el nombre del estado (`q`, `p`, `f`).

   - El rectángulo tiene coordenadas predefinidas y un color llamativo (`cyan`) para destacar visualmente.
   - La etiqueta en el centro del rectángulo muestra el estado actual.

3. **Dibujar la cadena restante:** La cadena restante que aún debe procesar el autómata se dibuja como texto en la parte inferior del estado. Si la cadena restante está vacía, se muestra el símbolo ε para indicar la transición al final.

4. **Dibujar la pila:** La pila se dibuja como una secuencia de rectángulos en orden descendente, donde cada rectángulo representa un elemento de la pila.

   - Cada rectángulo tiene dimensiones uniformes y está etiquetado con el símbolo que contiene (`X` o Z_0).
   - El rectángulo superior representa el tope de la pila, facilitando la identificación visual del elemento activo.

5. **Control del flujo de animación:** Si hay más transiciones por mostrar, el método se programa para ejecutarse nuevamente después de 1000 ms (1 segundo) utilizando `self.after(1000, self.dibujarTransicion)`. Si no hay más transiciones, se muestra un mensaje de finalización.

Flujo de ejecución del método La animación inicia automáticamente al crear una instancia de `AutomataAnimado`. El flujo completo se desarrolla de la siguiente manera:

1. Se carga la lista de transiciones generadas durante el procesamiento de la cadena.

2. La ventana gráfica se actualiza con cada transición, mostrando:

- El estado actual.
- La cadena restante.
- El contenido de la pila.

3. Una vez que se completan todas las transiciones, se informa al usuario mediante un mensaje visual.

2.2.4. Funcionamiento del Autómata

El autómata tiene los siguientes estados:

1. q**:** Estado inicial, donde se apilan X por cada 0 leído.
2. p**:** Estado intermedio, donde se desapilan X por cada 1 leído.
3. f**:** Estado final, alcanzado si la pila contiene únicamente Z_0 al finalizar la cadena.

La pila se inicializa con Z_0, un marcador especial que indica el fondo de la pila. Las transiciones se definen como:

- Leer 0 en q: Apilar X.
- Leer 1 en q: Cambiar a p y desapilar X.
- Leer 1 en p: Continuar en p y desapilar X.

Ejemplo: En el caso de una cadena válida como `0011`, la animación mostraría los siguientes pasos:

- **Paso 1:** Estado q, cadena restante `0011`, pila $[Z_0]$.
- **Paso 2:** Estado q, cadena restante `011`, pila $[X, Z_0]$.
- **Paso 3:** Estado p, cadena restante `11`, pila $[X, X, Z_0]$.
- **Paso 4:** Estado p, cadena restante `1`, pila $[X, Z_0]$.
- **Paso 5:** Estado p, cadena restante ε, pila $[Z_0]$.
- **Paso 6:** Estado f, cadena restante ε, pila $[Z_0]$.

2.3. Código Fuente

```
import random
import tkinter as tk  # Importa Tkinter para la animaci n gr fica

class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.next = None

class Pila:
    def __init__(self):
        self.top = None

    def isEmpty(self):
        return self.top is None

    def push(self, dato):
        nuevoNodo = Nodo(dato)
        nuevoNodo.next = self.top
        self.top = nuevoNodo

    def pop(self):
        if self.isEmpty():
            return None
        else:
            dato = self.top.dato
            self.top = self.top.next
            return dato

    def mostrarPila(self):
        """ Devuelve la pila completa como una lista de elementos desde el tope al fondo. """
        elementos = []
        actual = self.top
        while actual:
            elementos.append(actual.dato)
            actual = actual.next
        return elementos  # Orden de tope a fondo

    def peek(self):
        return None if self.isEmpty() else self.top.dato

class AutomataAnimado(tk.Tk):
    def __init__(self, cadena, transiciones):
        super().__init__()
        self.cadena = cadena
        self.transiciones = transiciones
```

```
46          self.title("Animaci n del Aut mata de Pila")
47          self.geometry("300x400")
48          self.canvas = tk.Canvas(self, width=300, height=400,
    bg="lightblue")
49          self.canvas.pack()
50          self.paso = 0
51          self.dibujarTransicion()
52
53      def dibujarTransicion(self):
54          if self.paso < len(self.transiciones):
55              estado, cadena_restante, pila = self.transiciones
    [self.paso]
56              self.canvas.delete("all")
57
58              # Dibuja el estado actual
59              self.canvas.create_rectangle(100, 50, 200, 100,
    fill="cyan", outline="black")
60              self.canvas.create_text(150, 75, text=f"Estado: {
    estado}", font=("Arial", 12))
61
62              # Dibuja la cadena restante
63              self.canvas.create_text(150, 150, text=f"Cadena:
    {cadena_restante}", font=("Arial", 12))
64
65              # Dibuja la pila (tope arriba)
66              y_pos = 200
67              for elemento in pila:  # Elementos en orden tope
    -> fondo
68                  self.canvas.create_rectangle(120, y_pos, 180,
     y_pos + 30, fill="white", outline="black")
69                  self.canvas.create_text(150, y_pos + 15, text
    =elemento, font=("Arial", 12))
70                  y_pos += 35
71
72              self.paso += 1
73              self.after(1000, self.dibujarTransicion)
74          else:
75              self.canvas.create_text(150, 350, text="
    Simulaci n finalizada", font=("Arial", 12), fill="green")
76
77
78  def procesar(cadena, estadoactual):
79      pilaAutomata = Pila()
80      pilaAutomata.push("Z0")  # Inicializamos la pila con Z0
81      indice = 0
82      transiciones = [(estadoactual, cadena, pilaAutomata.
    mostrarPila())]
83
84      with open("IDs.txt", "w", encoding="utf-8") as archivo:
```

```
85          archivo.write(f"(q, {cadena if cadena else ’  ’}, Z0)
    \n")
86
87          while indice < len(cadena):
88              simbolo = cadena[indice]
89              if simbolo not in ["0", "1"]:
90                  print("Cadena inv lida.")
91                  transiciones.append((estadoactual, cadena[
    indice:], pilaAutomata.mostrarPila()))
92                  return False, transiciones
93
94              if estadoactual == "q":
95                  if simbolo == "0":
96                      pilaAutomata.push("X")
97                  elif simbolo == "1":
98                      if pilaAutomata.peek() == "X":
99                          pilaAutomata.pop()
100                         estadoactual = "p"
101                     else:
102                         transiciones.append((estadoactual,
    cadena[indice:], pilaAutomata.mostrarPila()))
103                         return False, transiciones
104
105             elif estadoactual == "p":
106                 if simbolo == "1":
107                     if pilaAutomata.peek() == "X":
108                         pilaAutomata.pop()
109                     else:
110                         transiciones.append((estadoactual,
    cadena[indice:], pilaAutomata.mostrarPila()))
111                         return False, transiciones
112                 else:
113                     transiciones.append((estadoactual, cadena
    [indice:], pilaAutomata.mostrarPila()))
114                     return False, transiciones
115
116             indice += 1
117             tripleta = (estadoactual, cadena[indice:] if
    indice < len(cadena) else "  ", pilaAutomata.mostrarPila()
    )
118             transiciones.append(tripleta)
119
120             archivo.write(f"   ({estadoactual}, {tripleta
    [1]}, {’’.join(tripleta[2])})\n")
121
122         # Verificaci n final de aceptaci n
123         if estadoactual == "p" and pilaAutomata.peek() == "Z0
    ":
124             estadoactual = "f"
```

```
125             transiciones.append((estadoactual, "  ", ["Z0"]))
126             archivo.write(f"   (f,   , Z0)\n")
127             return True, transiciones
128         else:
129             transiciones.append((estadoactual, "  ",
    pilaAutomata.mostrarPila()))
130             return False, transiciones
131
132 def main():
133     print("1. Ingresar cadena\n2. Generar cadena")
134     opc = input("Seleccione una opci n: ")
135     if opc == "1":
136         cadena = input("Ingrese la cadena a analizar: ")
137     elif opc == "2":
138         cadena = "".join([str(random.randint(0, 1)) for _ in
    range(random.randint(1, 10))])
139         print(f"Cadena generada: {cadena}")
140     else:
141         print("Opci n no v lida.")
142         return
143
144     # Procesar la cadena
145     aceptada, transiciones = procesar(cadena, "q")
146
147     # Mostrar si la cadena pertenece al lenguaje
148     if aceptada:
149         print("La cadena pertenece al lenguaje {0^n 1^n | n
    >= 1}.")
150     else:
151         print("La cadena no pertenece al lenguaje {0^n 1^n |
    n >= 1}.")
152
153     # Mostrar la animaci n siempre si la cadena tiene <= 10
    caracteres
154     if len(cadena) <= 10:
155         app = AutomataAnimado(cadena, transiciones)
156         app.mainloop()
157
158 if __name__ == "__main__":
159     main()
```

Listing 2: Código Fuente del Programa 2

2.4. Ejecución

A continuación, se describe el proceso de ejecución y los resultados observados.

2.4.1. Modo Manual

En este modo, el usuario ingresa manualmente una cadena. El programa evalúa si pertenece al lenguaje definido, mostrando un resultado final y una animación gráfica de las transiciones.

2.4.2. Modo Automático

En este modo, el programa genera una cadena aleatoria para su evaluación, proporcionando resultados similares al modo manual, pero sin intervención del usuario.

2.4.3. Ejemplo de salida

```
1. Ingresar cadena
2. Generar cadena
Seleccione una opción: 1
Ingrese la cadena a analizar: 0011
La cadena pertenece al lenguaje {0^n 1^n | n >= 1}.
```

2.4.4. Archivo Generado

El programa guarda las transiciones de las descripciones instantáneas (ID's) en un archivo llamado `IDs.txt`. Cada línea del archivo representa una configuración del autómata en formato:

$$(q, \text{cadena\_restante}, \text{pila})$$

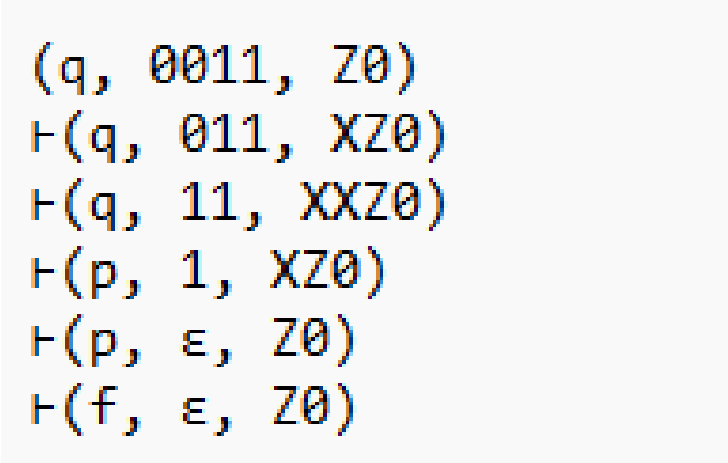

Figura 3: Ejemplo del contenido de `IDs.txt`.

2.4.5. Animación Gráfica

La animación muestra cada transición del autómata:

1. Estado actual.
2. Cadena restante.
3. Contenido de la pila.

Cada transición se actualiza visualmente en intervalos de 1 segundo, permitiendo observar el flujo de operación del autómata.

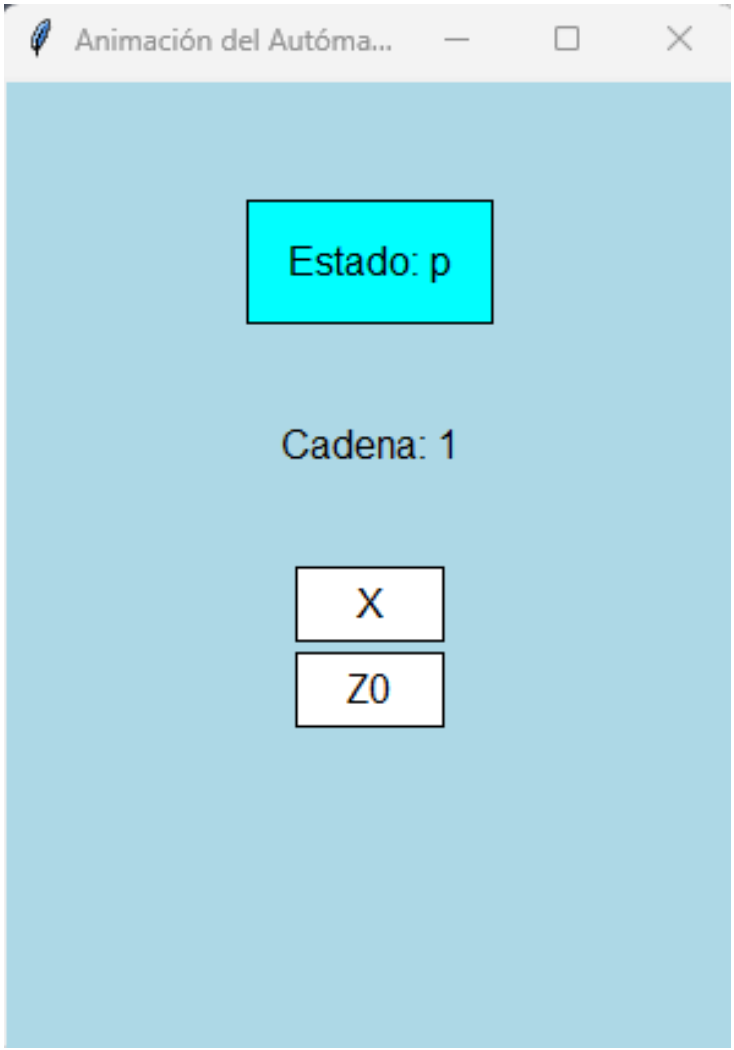

Figura 4: Captura de pantalla de la animación del autómata de pila.

3. Programa 3: Backus-Naur condicional IF

3.1. Introducción

El programa `Backus-Naur condicional IF` utiliza la notación Backus-Naur (BNF) para generar estructuras sintácticas de condicionales `if` de forma automática. Esta notación es ampliamente usada en la descripción de lenguajes formales y la generación de gramáticas para compiladores.

El objetivo del programa es derivar estructuras válidas de condicionales utilizando las reglas de una gramática en BNF. Posteriormente, estas derivaciones se traducen a pseudocódigo que simula la estructura generada. Este enfoque permite explorar la generación de estructuras complejas a partir de reglas simples, y tiene aplicaciones en áreas como diseño de lenguajes, testing de compiladores, y análisis sintáctico.

3.2. Descripción del Programa

El programa realiza los siguientes pasos principales:

1. Solicita al usuario la cantidad de condicionales `if` a generar, o selecciona un número aleatorio si el usuario lo permite.
2. Utiliza las reglas de la gramática BNF para generar derivaciones, que se registran en un archivo llamado `derivaciones.txt`.
3. Transforma la derivación final en pseudocódigo equivalente, que se guarda en el archivo `pseudocodigo.txt`.

3.2.1. Gramática en BNF

La gramática utilizada en el programa es la siguiente:

$$\text{S} \rightarrow \text{iCtSA}$$

$$\text{A} \rightarrow \text{eS} \quad | \quad \varepsilon$$

Donde:

- S: Es el símbolo inicial que representa la estructura general de las derivaciones.
- A: Representa una opción condicional adicional (`else`).
- C: Representa una condición en un `if`.
- ε: Representa la producción vacía (ausencia de una derivación).

3.2.2. Estructura del programa

El programa está compuesto por los siguientes elementos clave:

- **Generación de derivaciones:** Utiliza un algoritmo aleatorio para aplicar las reglas de producción de la gramática y generar derivaciones válidas.
- **Sustituciones:** Las variables S y A se reemplazan de acuerdo con las reglas definidas hasta que se alcanza la forma final.
- **Transformación a pseudocódigo:** Una vez obtenida la derivación final, se traduce a pseudocódigo en un formato estructurado y legible.

3.2.3. Flujo del Programa

1. **Selección del número de `if`:** El usuario ingresa el número de condicionales a generar, o el programa selecciona uno aleatoriamente entre 1 y 1000.
2. **Inicialización:** Se comienza con el símbolo inicial S y se escriben las derivaciones en `derivaciones.txt`.
3. **Aplicación de reglas:** De manera iterativa, se selecciona una variable (S o A) y se aplica una de las reglas de la gramática hasta completar la derivación.
4. **Generación de pseudocódigo:** Se reemplazan las producciones finales por el pseudocódigo correspondiente, y se guarda en `pseudocodigo.txt`.

3.3. Funcionamiento Detallado

El programa sigue un proceso iterativo y estructurado para generar derivaciones válidas basadas en la gramática BNF, con un enfoque claro en cómo se implementan cada una de las reglas y transformaciones. A continuación, se explica cada paso de manera más profunda:

Selección del número de condicionales

1. El programa comienza solicitando al usuario una opción:
    - **Modo manual:** El usuario ingresa el número de condicionales `if` que desea generar. Este valor debe estar entre 0 y 1000 para evitar sobrecarga.

- **Modo automático:** El programa selecciona un número aleatorio de condicionales entre 1 y 1000 usando la función `random.randint`.

2. Una vez seleccionado el número, este se almacena en la variable `ifs`, y el programa imprime esta cantidad para confirmar la entrada del usuario o la decisión automática.

Inicialización de variables

1. La derivación inicial se define como el símbolo inicial de la gramática, S, representado como una cadena.

2. Se inicializan contadores para rastrear el progreso:

   - `contadorIf`: Lleva un conteo de cuántos condicionales `if` se han generado.
   - `contadorDeriv`: Lleva un conteo del número total de derivaciones realizadas.

3. El archivo `derivaciones.txt` se abre en modo escritura, y la derivación inicial (S) se registra como el primer paso.

Generación de derivaciones El proceso de derivación se realiza en un ciclo que continúa hasta que se alcanza el número deseado de condicionales o se llega a un límite máximo de 1000 derivaciones para evitar bucles infinitos. Cada iteración del ciclo sigue los pasos siguientes:

1. **Selección de variable a derivar:**

   - Si aún no se han generado condicionales, se selecciona la variable S como punto de partida.
   - Una vez generado al menos un condicional, se selecciona aleatoriamente entre S y A, utilizando la función `random.choice`.

2. **Identificación de posiciones en la derivación:**

   - El programa busca todas las posiciones donde aparece la variable seleccionada dentro de la derivación actual.
   - Para encontrar estas posiciones, se utiliza un ciclo que recorre la cadena, identificando las apariciones de la variable seleccionada y almacenando sus índices en una lista.

- Si la variable no se encuentra, el programa vuelve a seleccionar S como variable para asegurar que el proceso continúe.

3. **Aplicación de reglas de producción:**
   - Una vez seleccionada una posición específica de la variable, se aplica la regla de producción correspondiente:
     - Si la variable es S, se expande a iCtSA, incrementando el contador `contadorIf`.
     - Si la variable es A, se expande a eS o se elimina (ε), dependiendo del contexto.
   - La derivación actual se actualiza reemplazando la variable en la posición seleccionada por su expansión.
4. **Registro de derivaciones:**
   - La derivación actualizada se escribe en el archivo `derivaciones.txt`, precedida por el operador $\Rightarrow$, indicando una nueva derivación.
   - El programa imprime la derivación actual en la consola para que el usuario pueda seguir el progreso.

Reemplazo final de A por ε Cuando se han completado todas las derivaciones necesarias, el programa realiza un paso final para terminar la derivación:

- Todas las apariciones restantes de A en la derivación se reemplazan por ε, eliminando cualquier elemento que no haya sido derivado.
- La derivación final se escribe en el archivo `derivaciones.txt`, precedida por el operador $\Rightarrow^*$ para denotar el estado final al que se llega después de cero o más derivaciones.

Transformación a pseudocódigo La derivación final se procesa para generar pseudocódigo, utilizando un enfoque basado en reglas:

1. Cada símbolo en la derivación final se traduce según la tabla de traducción definida en la gramática.
2. Para gestionar el formato del pseudocódigo:
   - Se utiliza un contador de tabulaciones (`contadorTabs`) para ajustar correctamente la indentación.

- Los bloques abiertos por t (`{` en el pseudocódigo) incrementan el contador de tabulaciones.
- Los bloques cerrados por S (`}` en el pseudocódigo) decrementan el contador.

3. El pseudocódigo resultante se guarda en el archivo `pseudocodigo.txt`.

Manejo de límites y errores Para garantizar la robustez del programa:

- Se impone un límite de 1000 derivaciones para evitar bucles infinitos en casos de gramáticas problemáticas.
- Las variables que no pueden ser derivadas se manejan reiniciando el proceso de selección de variables.
- El programa valida las entradas del usuario para asegurar que sean numéricas y estén dentro de los rangos permitidos.

3.4. Código Fuente

```
import random

def main():
    print("1. Ingresar cantidad de Ifs\n2. Dejar que la
    m quina decida")
    opc = input("Seleccione una opci n: ")
    if opc == "1":
        while True:
            ifs = int(input("Ingrese la cantidad de Ifs a
    realizar:"))
            if ifs >= 0 and ifs <= 1000:
                break
    elif opc == "2":
        ifs = random.randint(1, 1000)

    print(f"Ifs a realizar: {ifs}")


    contadorIf = 0
    contadorDeriv = 0
    derivaciones = "S"
    with open("derivaciones.txt", "w", encoding="utf-8") as
    archivo:
        archivo.write(f"S\n")
    print(derivaciones)

```

```
24
25      while contadorIf != ifs and contadorDeriv < 1000:
26          if contadorIf == 0:
27              variable = "S"
28          else:
29              variable = random.choice(["S", "A"])
30
31          indice = -1
32          indices = []
33          while True:
34              indice = derivaciones.find(variable, indice + 1)
        # Buscar desde el  ndice  siguiente
35              if indice == -1:  # Si no se encuentra m s
    ocurrencias, termina
36                  if indices == []:
37                      variable = "S"
38                      continue
39                  else:
40                      break
41              indices.append(indice)
42
43          indice = random.choice(indices)
44
45          if variable == "S":
46              derivaciones = derivaciones[:indice] + "iCtSA" +
    derivaciones[indice + 1:]
47              contadorIf += 1
48              contadorDeriv += 1
49          elif variable == "A":
50              if (indice + 1) < len(derivaciones):
51                  if derivaciones[indice + 1] == "A":
52                      derivaciones = derivaciones[:indice] + ""
     + derivaciones[indice + 1:]
53              else:
54                  derivaciones = derivaciones[:indice] + random
    .choice(["eS", ""]) + derivaciones[indice + 1:]
55              contadorDeriv += 1
56
57          print(f"Variable a derivar: {variable}")
58          print(derivaciones)
59          with open("derivaciones.txt", "a", encoding="utf-8")
    as archivo:
60              archivo.write(f"=> {derivaciones}\n")
61
62      #Reemplazar las A restantes por epsilon
63      derivaciones = derivaciones.replace("A", "")
64      with open("derivaciones.txt", "a", encoding="utf-8") as
    archivo:
65              archivo.write(f"=>* {derivaciones}\n")
```

```
66
67
68      with open("pseudocodigo.txt", "w", encoding="utf-8") as
    archivo:
69                  archivo.write("")
70
71      contadorTabs = 0
72      pseudocodigo = []  # Lista para guardar el pseudoc digo
73      for i in derivaciones:
74          line = ""
75          if i == "i":
76              line = "\t" * contadorTabs + "if"
77          elif i == "C":
78              line = "(condition)"
79          elif i == "t":
80              line = "{\n"
81              contadorTabs += 1
82          elif i == "S":
83              contadorTabs -= 1
84              line = "\t" * (contadorTabs + 1) + "Statement\n"
    + "\t" * contadorTabs + "}\n"
85          elif i == "e":
86              line = "\t" * contadorTabs + "else{\n"
87              contadorTabs += 1
88          pseudocodigo.append(line)
89      for i in range(contadorTabs):
90          contadorTabs -= 1
91          pseudocodigo.append("\t" * contadorTabs + "}\n")
92
93      with open("pseudocodigo.txt", "w", encoding="utf-8") as
    archivo:
94          archivo.writelines(pseudocodigo)
95
96
97
98
99  if __name__ == "__main__":
100     main()
```

Listing 3: Código Fuente del Programa 3

3.5. Ejecución

A continuación, se describe el proceso de ejecución y los resultados observados.

3.5.1. Modo Manual

En este modo, el usuario ingresa el número de condicionales que desea generar. El programa genera las derivaciones y produce el pseudocódigo correspondiente.

3.5.2. Modo Automático

En este modo, el programa selecciona aleatoriamente el número de condicionales entre 1 y 1000, generando las derivaciones y el pseudocódigo sin intervención del usuario.

3.5.3. Ejemplo de Salida

```
1. Ingresar cantidad de Ifs
2. Dejar que la máquina decida
Seleccione una opción: 1
Ingrese la cantidad de Ifs a realizar:10
Ifs a realizar: 10
S
Variable a derivar: S
iCtSA
Variable a derivar: S
iCtiCtSAA
Variable a derivar: S
iCtiCtiCtSAAA
Variable a derivar: A
iCtiCtiCtSAA
Variable a derivar: S
iCtiCtiCtiCtSAAA
Variable a derivar: A
iCtiCtiCtiCtSAA
Variable a derivar: S
iCtiCtiCtiCtiCtSAAA
Variable a derivar: A
iCtiCtiCtiCtiCtSAA
Variable a derivar: A
iCtiCtiCtiCtiCtSA
Variable a derivar: A
iCtiCtiCtiCtiCtS
Variable a derivar: S
iCtiCtiCtiCtiCtiCtSA
```

```
Variable a derivar: A
iCtiCtiCtiCtiCtiCtSeS
Variable a derivar: S
iCtiCtiCtiCtiCtiCtiCtSAeS
Variable a derivar: A
iCtiCtiCtiCtiCtiCtiCtSAeS
Variable a derivar: S
iCtiCtiCtiCtiCtiCtiCtiCtSAAeS
Variable a derivar: A
iCtiCtiCtiCtiCtiCtiCtiCtSAeS
Variable a derivar: S
iCtiCtiCtiCtiCtiCtiCtiCtSAeiCtSA
Variable a derivar: A
iCtiCtiCtiCtiCtiCtiCtiCtSAeiCtSeS
Variable a derivar: A
iCtiCtiCtiCtiCtiCtiCtiCtSAeiCtSeS
Variable a derivar: S
iCtiCtiCtiCtiCtiCtiCtiCtSAeiCtSeiCtSA
```

3.5.4. Archivos Generados

- `derivaciones.txt`: Contiene la secuencia de derivaciones generadas durante la ejecución.
- `pseudocodigo.txt`: Contiene el pseudocódigo equivalente a la derivación final.

```
S
=> iCtSA
=> iCtiCtSAA
=> iCtiCtiCtSAAA
=> iCtiCtiCtSAA
=> iCtiCtiCtiCtSAAA
=> iCtiCtiCtiCtSAA
=> iCtiCtiCtiCtiCtSAAA
=> iCtiCtiCtiCtiCtSAA
=> iCtiCtiCtiCtiCtSA
=> iCtiCtiCtiCtiCtS
=> iCtiCtiCtiCtiCtiCtSA
=> iCtiCtiCtiCtiCtiCtSeS
=> iCtiCtiCtiCtiCtiCtiCtSAeS
=> iCtiCtiCtiCtiCtiCtiCtSAeS
=> iCtiCtiCtiCtiCtiCtiCtiCtSAAeS
=> iCtiCtiCtiCtiCtiCtiCtiCtSAeS
=> iCtiCtiCtiCtiCtiCtiCtiCtSAeiCtSA
=> iCtiCtiCtiCtiCtiCtiCtiCtSAeiCtSeS
=> iCtiCtiCtiCtiCtiCtiCtiCtSAeiCtSeS
=> iCtiCtiCtiCtiCtiCtiCtiCtSAeiCtSeiCtSA
=>* iCtiCtiCtiCtiCtiCtiCtiCtSeiCtSeiCtS
```

Figura 5: Contenido del archivo `derivaciones.txt`.

```
if(condition){
        if(condition){
                if(condition){
                        if(condition){
                                if(condition){
                                        if(condition){
                                                if(condition){
                                                        Statement
                                                }
                                                else{
                                                        if(condition){
                                                                Statement
                                                        }
                                                        else{
                                                                if(condition){
                                                                        Statement
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }
}
```

Figura 6: Contenido del archivo `pseudocodigo.txt`.

4. Programa 4: Máquina de Turing

4.1. Introducción

La Máquina de Turing es un modelo computacional fundamental en la teoría de la computación. Este programa implementa una Máquina de Turing que reconoce el lenguaje $\{0^n1^n \mid n \geq 1\}$, donde el número de ceros debe ser igual al número de unos, y todos los ceros deben preceder a los unos.

Este lenguaje es un caso clásico en teoría de lenguajes formales, ya que no puede ser reconocido por un autómata finito ni por un autómata de pila. La Máquina de Turing resuelve este problema mediante el uso de una cinta infinita y una cabeza lectora/escritora, lo que le permite manipular y verificar la relación entre los ceros y unos en la cadena.

4.2. Descripción del Programa

El programa simula el funcionamiento de una Máquina de Turing, procesando una cadena de entrada y determinando si pertenece al lenguaje definido. Además, incluye una representación gráfica animada de la cinta y las transiciones de estados, así como un grafo del autómata basado en las transiciones definidas.

4.2.1. Elementos del Modelo

La Máquina de Turing está definida por los siguientes componentes:

- **Estados:** Un conjunto de estados, incluidos el estado inicial (q_0) y el estado de aceptación (q_4).
- **Cinta:** Una lista que representa la cinta infinita, donde cada celda contiene un símbolo.
- **Transiciones:** Una tabla que define las acciones a realizar dependiendo del estado actual y el símbolo leído.
- **Cabeza lectora/escritora:** Una posición en la cinta que puede leer y escribir símbolos, así como moverse a la izquierda (L) o derecha (R).

4.2.2. Tabla de Transiciones

La tabla de transiciones para este programa se define de la siguiente manera:

Estado	Símbolo 0	Símbolo 1	Símbolo X	Símbolo Y	Símbolo B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Cuadro 2: Tabla de transiciones de la Máquina de Turing.

Descripción de las transiciones:

- En q_0, si se encuentra un 0, se reemplaza por X y se avanza a la derecha. Si se encuentra Y, se transita directamente a q_3.
- En q_1, se avanza sobre los ceros y se reemplazan los unos por Y, retrocediendo luego a la izquierda.
- En q_2, se retrocede sobre los ceros y los X, regresando a q_0 al encontrar un X.
- En q_3, se avanza sobre los Y, y al encontrar un espacio vacío (B), se transita al estado de aceptación q_4.

4.2.3. Descripciones Instantáneas (IDs) de la Máquina de Turing

Una *descripción instantánea* (ID) en una Máquina de Turing es una representación concisa del estado actual del sistema durante su ejecución. Esto incluye la posición de la cabeza lectora, el estado actual de la máquina, y el contenido relevante de la cinta. El concepto de IDs permite estudiar y analizar las configuraciones intermedias de la máquina sin representar todo el espacio potencialmente infinito de la cinta. A continuación, se describe cómo el programa implementa este concepto teórico.

Definición Teórica de una ID Una ID para una Máquina de Turing se representa como:

$$X_1 X_2 \ldots X_{i-1} q X_i X_{i+1} \ldots X_n,$$

donde:

- q es el estado actual de la máquina.
- X_i es el símbolo que está siendo leído por la cabeza en la posición actual.

- $X_1, \ldots, X_{i-1}$ representan el contenido de la cinta a la izquierda de la cabeza.
- $X_{i+1}, \ldots, X_n$ representan el contenido de la cinta a la derecha de la cabeza.

Bajo ciertas condiciones especiales, si la cabeza está al borde izquierdo o derecho de las celdas no en blanco, los espacios en blanco de la cinta se representan explícitamente como B, asegurando que solo se incluyan celdas significativas.

Cómo se Imprimen las IDs en el Programa En el programa, las IDs se imprimen dinámicamente durante la ejecución de la máquina utilizando el método `paso()` de la clase `MaquinaDeTuring`. A continuación, se describe cómo se genera cada descripción instantánea:

1. **Obtención del estado actual:** La variable `self.estado_actual` mantiene el estado de la máquina en el paso actual.
2. **Contenido de la cinta:** Se extrae la secuencia de símbolos relevantes de la cinta mediante la lista `self.cinta`.
3. **Posición de la cabeza:** La posición de la cabeza lectora está almacenada en `self.posicion_cabeza`. Esto permite identificar qué símbolo está siendo leído en el momento.
4. **Formato de la ID:** La ID se representa en el archivo `IDs_MT.txt` utilizando el formato:

$$\mathtt{X}_1 \ldots \mathtt{X}_{i-1}(\mathtt{q})\mathtt{X}_i \ldots \mathtt{X}_n,$$

donde:

- Los símbolos a la izquierda y derecha de la cabeza se concatenan con el estado actual colocado entre paréntesis para indicar el punto de lectura.
- La cinta se actualiza dinámicamente después de cada transición.

Relación con el Modelo Teórico El programa implementa fielmente la definición teórica de IDs para las Máquinas de Turing:

- Representa explícitamente los estados y las posiciones de la cabeza lectora.

- Almacena y actualiza dinámicamente el contenido de la cinta y las posiciones significativas.
- Asegura que todas las IDs se guarden en el archivo `IDs_MT.txt` para su posterior análisis, haciendo uso de un formato compatible con la notación matemática estándar.

4.2.4. Métodos del Programa

El programa incluye varios métodos clave que encapsulan las funcionalidades principales de la Máquina de Turing. A continuación, se describen en detalle:

`__init__()`: Constructor de la clase `MaquinaDeTuring` Este método inicializa los atributos y la configuración de la Máquina de Turing:

- `self.cinta`: Convierte la cadena de entrada en una lista para facilitar la manipulación de cada celda.
- `self.transiciones`: Un diccionario que define las transiciones del autómata, con claves (estado_actual, símbolo_actual) y valores (nuevo_estado, nuevo_símbol
- `self.estado_actual`: Almacena el estado actual de la máquina, inicializado con el estado inicial (q_0).
- `self.estado_aceptacion`: Define el estado de aceptación (q_4).
- `self.posicion_cabeza`: Indica la posición de la cabeza lectora/escritora en la cinta, inicializada en 0.
- `self.animar`: Determina si se debe mostrar la animación gráfica, activada solo para cadenas de longitud menor o igual a 10.
- `self.root y self.canvas`: Configuran la ventana y el lienzo para la animación utilizando `Tkinter`.

`paso()`: Ejecución de un paso de la máquina Este método realiza una transición de la Máquina de Turing según las reglas definidas:

1. Obtiene el símbolo actual bajo la cabeza lectora. Si la posición de la cabeza está fuera de los límites de la cinta, se considera un espacio vacío (B).

2. Busca en el diccionario de transiciones una regla que coincida con el estado actual y el símbolo leído. Si no encuentra una transición válida, la máquina se detiene.

3. Actualiza:

   - El símbolo de la celda actual.
   - La posición de la cabeza lectora/escritora (L para izquierda, R para derecha).
   - El estado actual.

4. Si la animación está activada, llama al método `mostrar_cinta()` para reflejar visualmente los cambios en la cinta.

5. Devuelve `True` si se ejecutó un paso con éxito, o `False` si no hay una transición válida.

`ejecutar()`: Ciclo principal de la máquina Este método ejecuta la Máquina de Turing hasta que alcanza el estado de aceptación o no encuentra transiciones válidas:

1. Llama repetidamente al método `paso()` para realizar cada transición.

2. Registra cada descripción instantánea (ID) en el archivo `IDs_MT.txt`.

3. Si no hay transiciones válidas, imprime un mensaje indicando que la máquina se detuvo y muestra la configuración final de la cinta.

4. Si se alcanza el estado de aceptación, imprime un mensaje de éxito y la configuración final de la cinta.

5. Si la animación está activada, cierra la ventana gráfica al finalizar.

`mostrar_cinta()`: Visualización de la cinta Este método muestra gráficamente la cinta y la posición de la cabeza:

1. Limpia el lienzo para actualizar la visualización.

2. Dibuja un rectángulo para cada celda de la cinta, etiquetado con el símbolo que contiene.

3. Resalta la celda bajo la cabeza lectora/escritora con un color distintivo (`lightblue`).

4. Muestra el estado actual de la máquina sobre la cinta.

5. Realiza una pausa de 1 segundo para permitir que el usuario observe la transición.

`graficar_automata()`: Representación gráfica del autómata Este método genera un diagrama de estados y transiciones del autómata utilizando `Tkinter`:

1. Define las posiciones de cada estado en el lienzo.

2. Dibuja un círculo para cada estado, diferenciando el estado de aceptación con un doble borde.

3. Traza líneas con flechas para representar las transiciones entre estados, etiquetándolas con:

   - El símbolo leído.
   - El símbolo escrito.
   - La dirección de movimiento (R o L).

4. Maneja bucles sobre el mismo estado (*self-loops*) mediante curvas para evitar superposiciones.

4.2.5. Flujo del Programa

El programa realiza los siguientes pasos:

1. **Entrada:** El usuario ingresa una cadena manualmente o permite que el programa genere una cadena aleatoria.

2. **Configuración inicial:** La cinta se inicializa con la cadena de entrada, el estado inicial es q_0, y la posición de la cabeza es la primera celda.

3. **Procesamiento:**

   - En cada paso, el programa evalúa el estado actual y el símbolo bajo la cabeza.
   - Si hay una transición definida para esta combinación, realiza las siguientes acciones:
     a) Cambia al nuevo estado definido por la transición.
     b) Escribe el símbolo correspondiente en la celda actual.
     c) Mueve la cabeza en la dirección especificada.

- Si no hay una transición válida, la máquina se detiene.

4. **Aceptación o rechazo:** Si la máquina alcanza el estado de aceptación q_4, se acepta la cadena; de lo contrario, se rechaza.

4.3. Funcionamiento Detallado

El funcionamiento de la Máquina de Turing en este programa se organiza en tres componentes principales: inicialización, procesamiento y visualización. A continuación, se describe cada componente en detalle:

4.3.1. Inicialización del Programa

El programa comienza solicitando al usuario que ingrese una cadena o eligiendo una cadena generada aleatoriamente. Durante esta etapa:

1. La cinta se inicializa como una lista de caracteres, donde cada celda corresponde a un símbolo de la cadena.
2. El estado inicial (q_0) se define como el estado actual de la máquina.
3. La posición de la cabeza lectora/escritora se establece en la primera celda de la cinta.
4. Se inicializa el archivo `IDs_MT.txt` para registrar las descripciones instantáneas (IDs).

4.3.2. Procesamiento Paso a Paso

El procesamiento se realiza mediante el método `paso`, que aplica las transiciones definidas en la tabla. Este método ejecuta las siguientes acciones en cada iteración:

1. **Lectura del símbolo actual:**
   - La cabeza lectora obtiene el símbolo de la celda actual. Si la posición de la cabeza excede los límites de la cinta, se interpreta como un espacio vacío (B).
2. **Búsqueda de transición:**
   - Se consulta la tabla de transiciones usando la clave (estado_actual, símbolo_actual).
   - Si no existe una transición válida, la máquina se detiene y la cadena se rechaza.

- Si existe una transición válida, se obtiene el nuevo estado, el símbolo a escribir y la dirección de movimiento.

3. **Escritura en la cinta:**
   - El símbolo en la celda actual se reemplaza por el nuevo símbolo definido en la transición.
   - Si la cabeza se encuentra en una posición fuera de los límites actuales de la cinta, se expande la cinta añadiendo un espacio vacío (B).
4. **Movimiento de la cabeza:**
   - La cabeza se mueve a la izquierda (L) o a la derecha (R) según lo indique la transición.
   - Si la dirección es L y la posición de la cabeza es 0, se añade un espacio vacío al inicio de la cinta para evitar errores.
5. **Cambio de estado:**
   - El estado actual de la máquina se actualiza al nuevo estado definido por la transición.
6. **Registro de IDs:**
   - El estado actual, la cinta y la posición de la cabeza se registran en el archivo `IDs_MT.txt`.

El ciclo continúa hasta que la máquina alcanza el estado de aceptación (q_4) o no existen transiciones válidas para la configuración actual.

4.3.3. Visualización de la Máquina

La animación gráfica de la cinta y las transiciones se realiza a través del método `mostrar_cinta`, que:

1. Limpia el lienzo (`Canvas`) para preparar la visualización de la cinta en su estado actual.
2. Dibuja cada celda de la cinta como un rectángulo, etiquetado con el símbolo que contiene.
3. Resalta la celda bajo la cabeza lectora/escritora con un color distintivo (`lightblue`).

4. Muestra el estado actual de la máquina sobre la cinta.

5. Realiza una pausa de 1 segundo para que el usuario observe los cambios antes de la siguiente transición.

4.3.4. Representación del Autómata

El método `graficar_automata` genera un diagrama del autómata, representando gráficamente los estados y sus transiciones.

4.3.5. Ejemplo de Ejecución Paso a Paso

Para la cadena 0011:

- **Paso 1:** Estado q_0, posición de la cabeza = 0, cinta = `0011B`.
  - Símbolo actual: 0.
  - Transición: $(q_0, 0) \rightarrow (q_1, X, R)$.
  - Resultado: q_1, cinta = `X011B`, cabeza se mueve a la derecha.
- **Paso 2:** Estado q_1, posición de la cabeza = 1, cinta = `X011B`.
  - Símbolo actual: 0.
  - Transición: $(q_1, 0) \rightarrow (q_1, 0, R)$.
  - Resultado: q_1, cinta sin cambios, cabeza se mueve a la derecha.
- **Paso 3:** Estado q_1, posición de la cabeza = 2, cinta = `X011B`.
  - Símbolo actual: 1.
  - Transición: $(q_1, 1) \rightarrow (q_2, Y, L)$.
  - Resultado: q_2, cinta = `X0Y1B`, cabeza se mueve a la izquierda.
- **Paso 4:** Estado q_2, posición de la cabeza = 1, cinta = `X0Y1B`.
  - Símbolo actual: 0.
  - Transición: $(q_2, 0) \rightarrow (q_2, 0, L)$.
  - Resultado: q_2, cinta sin cambios, cabeza se mueve a la izquierda.
- **Paso 5:** Estado q_2, posición de la cabeza = 0, cinta = `X0Y1B`.
  - Símbolo actual: X.
  - Transición: $(q_2, X) \rightarrow (q_0, X, R)$.

- Resultado: q_0, cinta sin cambios, cabeza se mueve a la derecha.

El proceso continúa hasta que la máquina alcanza el estado q_4, con la cinta final $XXYYB$, que indica que la cadena pertenece al lenguaje.

4.4. Código Fuente

```
import random
import tkinter as tk
import time
import math  # Para c lculos de  ngulos  y posiciones
from collections import defaultdict

class MaquinaDeTuring:
    def __init__(self, cinta, transiciones, estado_inicial, estado_aceptacion):
        self.cinta = list(cinta)  # Convertir la cinta en una lista para f cil manipulaci n
        self.transiciones = transiciones
        self.estado_actual = estado_inicial
        self.estado_aceptacion = estado_aceptacion
        self.posicion_cabeza = 0

        self.animar = (len(self.cinta) <= 10)  # Solo animar si la longitud <= 10
        if self.animar:
            self.root = tk.Tk()
            self.root.title("Animaci n de la M quina de Turing")
            self.canvas = tk.Canvas(self.root, width=600, height=200, bg="white")
            self.canvas.pack()
        else:
            self.root = None
            self.canvas = None

    def paso(self):
        # Obtener el s mbolo actual bajo la cabeza
        simbolo_actual = self.cinta[self.posicion_cabeza] if self.posicion_cabeza < len(self.cinta) else 'B'

        # Buscar la transici n correspondiente
        clave = (self.estado_actual, simbolo_actual)
        if clave not in self.transiciones:
            return False  # No hay transici n v lida, detenerse

```

```
34          nuevo_estado, nuevo_simbolo, direccion = self.
    transiciones[clave]
35
36          if self.posicion_cabeza < len(self.cinta):
37              self.cinta[self.posicion_cabeza] = nuevo_simbolo
38          else:
39              self.cinta.append(nuevo_simbolo)
40
41          if direccion == 'R':
42              self.posicion_cabeza += 1
43          elif direccion == 'L':
44              self.posicion_cabeza -= 1
45
46          self.estado_actual = nuevo_estado
47
48          if self.animar:
49              self.mostrar_cinta()
50
51          return True
52
53      def ejecutar(self):
54          while self.estado_actual != self.estado_aceptacion:
55              if not self.paso():
56                  print("M quina detenida: no hay transici n
    v lida.")
57                  print("Cinta final:", ''.join(self.cinta))
58
59                  if self.animar and self.root is not None:
60                      self.root.destroy()
61                  exit()
62
63              with open("IDs_MT.txt", "a", encoding="utf-8") as
     archivo:
64                  archivo.write(f"\ n  {''.join(self.cinta[:
    self.posicion_cabeza])}({self.estado_actual}){''.join(self
    .cinta[self.posicion_cabeza:])}")
65
66          print("Estado de aceptaci n alcanzado.")
67          print("Cinta final:", ''.join(self.cinta))
68
69          if self.animar and self.root is not None:
70              self.root.destroy()
71
72      def mostrar_cinta(self):
73          self.canvas.delete("all")
74
75          x_inicial = 50
76          y_inicial = 80
77          ancho_celda = 40
```

```
78          alto_celda = 40
79
80          for i, simbolo in enumerate(self.cinta):
81              x1 = x_inicial + i * ancho_celda
82              y1 = y_inicial
83              x2 = x1 + ancho_celda
84              y2 = y1 + alto_celda
85
86              if i == self.posicion_cabeza:
87                  color = "lightblue"
88              else:
89                  color = "white"
90
91              self.canvas.create_rectangle(x1, y1, x2, y2, fill
    =color, outline="black")
92              self.canvas.create_text((x1 + x2)/2, (y1 + y2)/2,
     text=simbolo, font=("Arial", 16))
93
94          self.canvas.create_text(300, 30, text=f"Estado: {self
    .estado_actual}", font=("Arial", 14), fill="blue")
95
96          self.root.update()
97          time.sleep(1)
98
99  def graficar_automata(transiciones):
100
101     ventana_automata = tk.Tk()
102     ventana_automata.title("Aut mata (Diagrama de estados)")
103
104     ancho_canvas = 800
105     alto_canvas = 400
106     canvas_auto = tk.Canvas(ventana_automata, width=
    ancho_canvas, height=alto_canvas, bg="white")
107     canvas_auto.pack()
108
109     # Posiciones fijas para los estados
110     posiciones_estados = {
111         'q0': (100, 200),
112         'q1': (250, 100),
113         'q2': (250, 300),
114         'q3': (400, 100),
115         'q4': (550, 200)
116     }
117
118     radios = 30
119
120     # Dibujamos los estados
121     for estado, (cx, cy) in posiciones_estados.items():
122         # Si es estado de aceptaci n, dibujamos doble  valo
```

```
123         if estado == 'q4':
124             canvas_auto.create_oval(cx - radios - 5, cy -
    radios - 5,
125                                     cx + radios + 5, cy +
    radios + 5,
126                                     outline="blue", width=2)
127         #   valo   base
128         canvas_auto.create_oval(cx - radios, cy - radios, cx
    + radios, cy + radios, fill="lightgray")
129         canvas_auto.create_text(cx, cy, text=estado, font=("
    Arial", 14, "bold"), fill="black")
130 
131     # Diccionario para contar cu ntos self-loops hay por
    estado
132     self_loop_count = defaultdict(int)
133 
134     def dibujar_flecha_arco(cx, cy, etiqueta, offset):
135         """
136         Dibuja un arco curvado (self-loop) que sale del borde
     del estado
137         y regresa al mismo sin superponerse con otros self-
    loops.
138         """
139         # Empezamos en el borde superior del c rculo
140         x_inicio = cx
141         y_inicio = cy - radios
142 
143         # Ajustamos el offset para separar las curvas
144         # Cada self-loop adicional har  que la curva se
    desplace
145         desplazamiento = 40 + offset * 40
146 
147         # Puntos de control
148         x_control1 = cx - (radios + desplazamiento)
149         y_control1 = y_inicio - 40 - offset * 10
150         x_control2 = cx + (radios + desplazamiento)
151         y_control2 = y_inicio - 40 - offset * 10
152 
153         x_fin = x_inicio
154         y_fin = y_inicio
155 
156         # Trazamos la l nea curva (spline) con flecha
157         puntos = [
158             x_inicio, y_inicio,
159             x_control1, y_control1,
160             x_control2, y_control2,
161             x_fin, y_fin
162         ]
163         canvas_auto.create_line(
```

```
164             *puntos,
165             fill="black",
166             width=2,
167             smooth=True,
168             arrow=tk.LAST
169         )
170 
171         # Etiqueta aproximadamente en el punto m s alto de
    la curva
172         xm = (x_control1 + x_control2) / 2
173         ym = (y_control1 + y_control2) / 2
174         canvas_auto.create_text(xm, ym - 15, text=etiqueta,
    font=("Arial", 10, "bold"), fill="blue")
175 
176     def dibujar_flecha(x1, y1, x2, y2, etiqueta):
177         """
178         Ajusta la l nea para que las flechas toquen el borde
     de los c rculos,
179         en lugar de sus centros, y coloca la etiqueta en el
    punto medio.
180         """
181         angulo = math.atan2(y2 - y1, x2 - x1)
182 
183         # Ajustamos para que la flecha parta desde el borde
    del c rculo de origen
184         x1r = x1 + radios * math.cos(angulo)
185         y1r = y1 + radios * math.sin(angulo)
186         # Ajustamos para que termine en el borde del c rculo
     de destino
187         x2r = x2 - radios * math.cos(angulo)
188         y2r = y2 - radios * math.sin(angulo)
189 
190         # Dibuja la flecha
191         canvas_auto.create_line(x1r, y1r, x2r, y2r, arrow=tk.
    LAST, width=2)
192 
193         # Punto medio para la etiqueta
194         xm = (x1r + x2r) / 2
195         ym = (y1r + y2r) / 2
196         canvas_auto.create_text(xm, ym - 10, text=etiqueta,
    font=("Arial", 10, "bold"), fill="blue")
197 
198     # Para reemplazar 'R' o 'L' por flechas reales (    o
    )
199     direccion_a_flecha = {'R': '    ', 'L': '    '}
200 
201     # Dibujamos las transiciones
202     for (estado_origen, simbolo_entrada), (estado_destino,
    simbolo_salida, direccion) in transiciones.items():
```

```
203         dir_flecha = direccion_a_flecha.get(direccion, '')
204         etiqueta = f"{simbolo_entrada}/{simbolo_salida}{
    dir_flecha}"
205 
206         if estado_origen == estado_destino:
207             # Self-loop
208             cx, cy = posiciones_estados[estado_origen]
209             offset = self_loop_count[estado_origen]
210             dibujar_flecha_arco(cx, cy, etiqueta, offset)
211             self_loop_count[estado_origen] += 1
212         else:
213             if (estado_origen in posiciones_estados) and (
    estado_destino in posiciones_estados):
214                 x1, y1 = posiciones_estados[estado_origen]
215                 x2, y2 = posiciones_estados[estado_destino]
216                 dibujar_flecha(x1, y1, x2, y2, etiqueta)
217 
218     ventana_automata.mainloop()
219 
220 def main():
221     print("1. Ingresar cadena\n2. Generar cadena")
222     opcion = input("Seleccione una opci n: ")
223 
224     if opcion == "1":
225         cadena = input("Ingrese la cadena a analizar: ")
226     elif opcion == "2":
227         cadena = "".join([str(random.randint(0, 1)) for _ in
    range(random.randint(1, 1000))])
228         print(f"Cadena generada: {cadena}")
229     else:
230         print("Opci n no v lida.")
231         return
232 
233     transiciones = {
234         ('q0', '0'): ('q1', 'X', 'R'),
235         ('q0', 'Y'): ('q3', 'Y', 'R'),
236         ('q1', '0'): ('q1', '0', 'R'),
237         ('q1', '1'): ('q2', 'Y', 'L'),
238         ('q1', 'Y'): ('q1', 'Y', 'R'),
239         ('q2', '0'): ('q2', '0', 'L'),
240         ('q2', 'X'): ('q0', 'X', 'R'),
241         ('q2', 'Y'): ('q2', 'Y', 'L'),
242         ('q3', 'Y'): ('q3', 'Y', 'R'),
243         ('q3', 'B'): ('q4', 'B', 'R'),
244     }
245 
246     graficar_automata(transiciones)
247 
248     estado_inicial = 'q0'
```

```
249     estado_aceptacion = 'q4'
250     maquina = MaquinaDeTuring(cadena, transiciones,
        estado_inicial, estado_aceptacion)
251
252     with open("IDs_MT.txt", "w", encoding="utf-8") as archivo
        :
253         archivo.write(f"(q0){cadena}")
254
255     maquina.ejecutar()
256
257 if __name__ == "__main__":
258     main()
```

Listing 4: Código Fuente del Programa 4: Máquina de Turing

4.5. Ejecución

A continuación, se presentan ejemplos de ejecución y resultados obtenidos.

```
1. Ingresar cadena
2. Generar cadena
Seleccione una opción: 1
Ingrese la cadena a analizar: 0011
Estado de aceptación alcanzado.
Cinta final: XXYYB
```

4.5.1. Gráfica del Autómata

El programa genera una representación gráfica del autómata basada en la tabla de transiciones, mostrando los estados y sus conexiones.

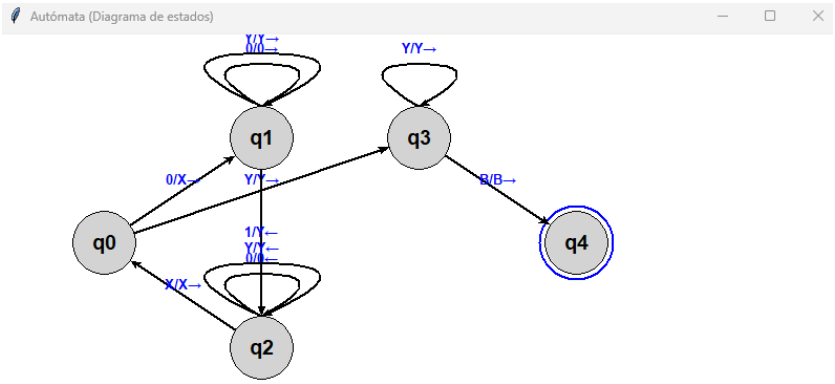

Figura 7: Gráfica del autómata para la Máquina de Turing.

4.5.2. Animación de la Máquina de Turing

El programa también genera una animación de la máquina de Turing basada en las descripciones instantáneas.

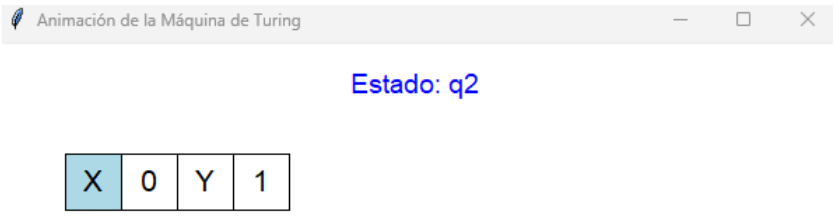

Figura 8: Gráfica del autómata para la Máquina de Turing.

4.5.3. IDs de la Máquina de Turing

Por último, se muestran dichas descripciones instantáneas en el archivo de texto.

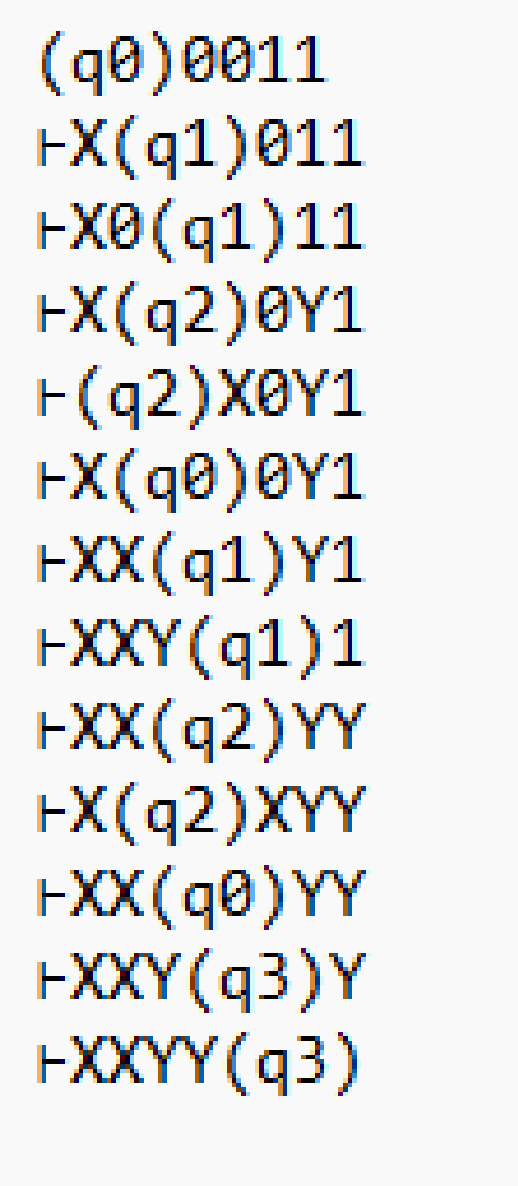

Figura 9: Descripciones instantáneas correspondientes a la ejecución.

5. Programa Principal: Coordinador de Ejecuciones

5.1. Introducción

El programa principal funciona como un coordinador que permite ejecutar los cuatro programas previamente desarrollados: el `Buscador de Palabras`, el `Autómata de Pila`, el `Generador Backus-Naur`, y la `Máquina de Turing`. Este programa proporciona dos modos de ejecución:

- **Modo Automático:** Ejecuta aleatoriamente uno de los programas en una secuencia de 10 iteraciones.
- **Modo Manual:** Permite al usuario seleccionar y ejecutar uno de los programas directamente.

5.2. Descripción del Programa

El programa está compuesto por un menú principal y dos funciones principales que gestionan la ejecución de los programas disponibles. A continuación, se explican los detalles de su funcionamiento.

5.2.1. Estructura del Menú Principal

El menú principal presenta al usuario tres opciones:

1. Ejecutar 10 programas aleatorios automáticamente.
2. Elegir un programa para ejecutar manualmente.
3. Salir del programa.

El programa emplea ciclos `while` para permitir múltiples interacciones con el menú, hasta que el usuario decida salir.

5.2.2. Programas Disponibles

El programa gestiona los siguientes módulos importados:

- `buscador.main`: Representa el *Buscador de Palabras*.
- `automatapila.main`: Representa el *Autómata de Pila*.
- `backusNaur.main`: Representa el *Generador Backus-Naur*.
- `MT.main`: Representa la *Máquina de Turing*.

5.2.3. Ejecución Automática

La función `ejecutar_programa_aleatorio` selecciona aleatoriamente un programa de la lista disponible y lo ejecuta. Esto se realiza mediante la biblioteca estándar `random`.

5.2.4. Ejecución Manual

La función `ejecutar_programa_manual` permite al usuario seleccionar un programa específico de la lista. El menú muestra todos los programas disponibles y una opción para salir. Si el usuario elige un programa válido, este se ejecuta inmediatamente.

5.3. Funcionamiento Detallado

A continuación se describe cómo el programa gestiona la ejecución:

5.3.1. `ejecutar_programa_aleatorio`

- Selecciona un programa de manera aleatoria utilizando `random.choice`.
- Imprime el nombre del programa seleccionado.
- Llama a la función principal (`main`) del programa correspondiente.

5.3.2. `ejecutar_programa_manual`

- Imprime una lista numerada de los programas disponibles.
- Solicita al usuario que seleccione un programa ingresando un número.
- Ejecuta el programa seleccionado si la opción es válida.
- Permite salir del menú manual seleccionando la última opción.

5.3.3. `main`

- Presenta un menú principal con tres opciones:
  1. Ejecutar 10 programas aleatorios.
  2. Elegir un programa manualmente.
  3. Salir.
- Controla las interacciones del usuario mediante ciclos `while`.
- Finaliza el programa si el usuario selecciona la opción de salir.

5.4. Código Fuente

```
import random
import time

import buscador
import automatapila
import backusNaur
import MT

programas = {
    "buscador": buscador.main,
    "automatapila": automatapila.main,
    "backusNaur": backusNaur.main,
    "MT": MT.main,
}

def ejecutar_programa_aleatorio():
    programa_seleccionado = random.choice(list(programas.keys
    ()))
    print(f"\nEjecutando: {programa_seleccionado}")
    programas[programa_seleccionado]()

def ejecutar_programa_manual():
    print("\nProgramas disponibles:")
    for i, programa in enumerate(programas.keys(), 1):
        print(f"{i}. {programa}")
    print(f"{len(programas) + 1}. Salir")

    while True:
        try:
            opcion = int(input("Seleccione un programa por su
     n mero: "))
            if 1 <= opcion <= len(programas):
                programa_seleccionado = list(programas.keys()
    )[opcion - 1]
                print(f"\nEjecutando: {programa_seleccionado}
    ")
                programas[programa_seleccionado]()
                break
            elif opcion == len(programas) + 1:
                print("Saliendo del men  manual...")
                break
            else:
                print("Opci n inv lida. Intente nuevamente.
    ")
        except ValueError:
            print("Entrada inv lida. Por favor, ingrese un
    n mero.")
```

```
42
43 def main():
44     while True:
45         print("\n--- MEN  PRINCIPAL ---")
46         print("1. Ejecutar 10 programas aleatorios
    autom ticamente")
47         print("2. Elegir un programa para ejecutar
    manualmente")
48         print("3. Salir")
49
50         try:
51             opcion = int(input("Seleccione una opci n: "))
52             if opcion == 1:
53                 iteraciones = 10
54                 print("\nEjecutando 10 programas de forma
    autom tica...")
55                 for _ in range(iteraciones):
56                     ejecutar_programa_aleatorio()
57                     time.sleep(random.uniform(1, 3))
58             elif opcion == 2:
59                 ejecutar_programa_manual()
60             elif opcion == 3:
61                 print("Saliendo del programa...")
62                 break
63             else:
64                 print("Opci n inv lida. Intente nuevamente.
    ")
65         except ValueError:
66             print("Entrada inv lida. Por favor, ingrese un
    n mero.")
67
68 if __name__ == "__main__":
69     main()
```

Listing 5: Código Fuente del Programa Principal

5.5. Ejecución

El programa permite gestionar la ejecución de los módulos individuales de manera ordenada y eficiente. Este diseño modular facilita la integración y pruebas de los programas individuales dentro de un sistema centralizado.A continuación, se presentan un ejemplo de ejecución.

```
--- MENÚ PRINCIPAL ---
1. Ejecutar 10 programas aleatorios automáticamente
2. Elegir un programa para ejecutar manualmente
3. Salir
```

```
Seleccione una opción: 2

Programas disponibles:
1. buscador
2. automatapila
3. backusNaur
4. MT
5. Salir
Seleccione un programa por su número: 3

Ejecutando: backusNaur
1. Ingresar cantidad de Ifs
2. Dejar que la máquina decida
Seleccione una opción: 1
Ingrese la cantidad de Ifs a realizar:10
Ifs a realizar: 10
S
Variable a derivar: S
iCtSA
Variable a derivar: A
iCtS
Variable a derivar: S
iCtiCtSA
Variable a derivar: S
iCtiCtiCtSAA
Variable a derivar: S
iCtiCtiCtiCtSAAA
Variable a derivar: S
iCtiCtiCtiCtiCtSAAAA
Variable a derivar: A
iCtiCtiCtiCtiCtSAAA
Variable a derivar: A
iCtiCtiCtiCtiCtSAA
Variable a derivar: S
iCtiCtiCtiCtiCtiCtSAAA
Variable a derivar: A
iCtiCtiCtiCtiCtiCtSAA
Variable a derivar: A
iCtiCtiCtiCtiCtiCtSA
Variable a derivar: S
iCtiCtiCtiCtiCtiCtiCtSAA
```

```
Variable a derivar: S
iCtiCtiCtiCtiCtiCtiCtSAAA
Variable a derivar: A
iCtiCtiCtiCtiCtiCtiCtSAA
Variable a derivar: S
iCtiCtiCtiCtiCtiCtiCtiCtSAAA
Variable a derivar: A
iCtiCtiCtiCtiCtiCtiCtiCtSAA
Variable a derivar: A
iCtiCtiCtiCtiCtiCtiCtiCtSA
Variable a derivar: A
iCtiCtiCtiCtiCtiCtiCtiCtSeS
Variable a derivar: S
iCtiCtiCtiCtiCtiCtiCtiCtSeiCtSA

--- MENÚ PRINCIPAL ---
1. Ejecutar 10 programas aleatorios automáticamente
2. Elegir un programa para ejecutar manualmente
3. Salir
Seleccione una opción: 3
Saliendo del programa...
```

6. Conclusión

El presente reporte documenta de manera exhaustiva la ejecución y el propósito de los cuatro programas principales (`Buscador de Palabras`, `Autómata de Pila`, `Generador Backus-Naur` y `Máquina de Turing`), así como la funcionalidad del programa coordinador que actúa como un núcleo central para su integración. Este conjunto de aplicaciones resalta no solo la modularidad de su diseño, sino también la capacidad de los programas para operar de forma autónoma o conjunta, según los requerimientos del usuario.

Cada programa implementa conceptos fundamentales de teoría de autómatas, gramáticas y lenguajes formales, ilustrando cómo estos principios pueden materializarse en soluciones prácticas. Desde la capacidad del `Buscador de Palabras` para analizar y procesar texto mediante un autómata finito determinista, hasta la `Máquina de Turing` que valida un lenguaje más complejo, todos los componentes demuestran la aplicabilidad de estos conceptos en problemas computacionales. Además, el `Autómata de Pila` y el `Generador Backus-Naur` destacan por su relevancia en el reconocimiento de lenguajes libres de contexto y la generación de sintaxis programática, respectivamente, consolidando su importancia en el diseño de compiladores y validación de lenguajes.

El diseño modular del programa coordinador es un elemento clave que merece destacar. Este núcleo principal permite a los usuarios interactuar con los programas de manera flexible mediante dos modos: el **modo automático**, que ejecuta una secuencia de programas seleccionados aleatoriamente, y el **modo manual**, que brinda control total sobre la ejecución de los módulos. Esta estructura no solo facilita la integración de los programas, sino que también asegura su escalabilidad, permitiendo incorporar nuevas funcionalidades o módulos adicionales sin comprometer la cohesión del sistema.

Adicionalmente, los resultados obtenidos durante las pruebas confirman que cada programa cumple con su propósito específico. El `Buscador de Palabras` demostró su precisión al identificar palabras clave dentro de un texto, incluso en escenarios complejos con grandes volúmenes de datos. El `Autómata de Pila` ilustró de manera visual cómo opera un modelo teórico para reconocer lenguajes balanceados, mientras que el `Generador Backus-Naur` automatizó la generación de estructuras condicionales en pseudocódigo. Por último, la `Máquina de Turing` mostró su potencia al procesar cadenas y alcanzar estados de aceptación según las reglas definidas, validando así su capacidad para manejar problemas computacionales complejos.

Desde el punto de vista pedagógico, este conjunto de programas no solo es una herramienta funcional, sino también un recurso didáctico para explorar y comprender conceptos teóricos avanzados. Las animaciones gráficas y las

representaciones visuales de los autómatas y máquinas utilizadas permiten a los usuarios (particularmente estudiantes y entusiastas de la computación) visualizar procesos abstractos de una manera más tangible y comprensible.

Por último, la implementación práctica de estos programas resalta la importancia de una planificación estructurada y un enfoque modular en el desarrollo de software. Cada módulo fue diseñado con objetivos claros, y su integración fue posible gracias a una arquitectura flexible y bien definida. Este proyecto demuestra cómo un enfoque basado en teoría de la computación puede traducirse en soluciones prácticas y funcionales, útiles tanto en contextos educativos como en aplicaciones del mundo real.

7. Referencias

Referencias

[1] Hopcroft, J. E., Motwani, R., Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

[2] Giró, J. R., Vázquez, G. A., Meloni, A. M., & Constable, J. (2016). *Lenguajes formales y teoría de autómatas*. Alfaomega Grupo Editor.

[3] Pardo, L. M. (2015). *Teoría de Autómatas y Lenguajes Formales*. Universidad de Cantabria.

[4] Santucho, M. I. (2015). *Teoría de Autómatas y Lenguajes Formales*.

[5] Universidad Internacional de Valencia. (2023). *Teoría de Autómatas y Lenguajes Formales*. Disponible en: `https://www.universidadviu.com/sites/universidadviu.com/files/media_files/13GIIN%20Guia%20Didactica_Teoria%20de%20Automatas%20y%20Lenguajes%20Formales_0.pdf`