Informe del Proyecto: Generación de Autómatas a partir de Expresiones Regulares

1. Introducción

El objetivo principal de este proyecto es implementar un generador de autómatas finitos a partir de expresiones regulares. Este generador procesa las expresiones regulares ingresadas por el usuario y construye un autómata finito que se puede visualizar como una imagen. Además, el proyecto incluye la validación de expresiones regulares, el manejo de transiciones de estado y la visualización del autómata resultante.

2. Descripción del Sistema

El sistema se basa en autómatas finitos deterministas, utilizados para procesar cadenas de texto y determinar si cumplen con condiciones definidas por una expresión regular.

El generador de autómatas toma una expresión regular como entrada y construye un autómata que representa su comportamiento. Este se construye paso a paso, agregando estados y transiciones entre ellos, para aceptar cadenas que coincidan con la expresión regular.

Además de su funcionalidad principal, el sistema ofrece una interfaz para visualizar los autómatas generados, guardándolos en formato PNG y permitiendo al usuario ver su estructura.

3. Estructura del Código

El código se divide en varios módulos para organizar las distintas funcionalidades del proyecto:

3.1 Módulo `world/util.py`

Este módulo contiene la clase `Util` con dos métodos principales:

- `matchStrings(expression, text)`: Busca las posiciones de coincidencia de una expresión regular en un texto dado.
- `removeSpaces(str)`: Elimina los espacios en blanco al inicio de una cadena.

3.2 Módulo `draw_automata/util.py`

Este módulo contiene las clases y métodos para crear y visualizar el autómata:

- **Clase** `State`: Representa un estado en el autómata, con un identificador único, un diccionario de transiciones y una bandera que indica si es un estado final.
- **Clase** `Automata`: Representa el autómata, con métodos para crear estados, agregar transiciones y gestionarlo.
- **Clase** `GeneratorAutomata`: Núcleo del generador de autómatas. Procesa una expresión regular, crea el autómata correspondiente, aplica transiciones y genera su visualización.
  - Métodos clave:
    - `validate_expresion(expr)`: Valida la sintaxis de la expresión regular.
    - `process_expresion(expr)`: Procesa la expresión regular y genera el autómata.
    - `visualize_automata()`: Genera una imagen del autómata usando Graphviz y la guarda como archivo PNG.

3.3 Módulo `automata/util.py`

Este módulo contiene la clase `Automata`, que simula el procesamiento de una cadena de entrada usando un autómata. Incluye métodos para verificar si una cadena es válida según las transiciones definidas y maneja excepciones personalizadas para caracteres inválidos y transiciones no válidas.

4. Funcionalidades Principales

El sistema ofrece las siguientes funcionalidades clave:

1. **Generación de Autómatas Finitos Deterministas**:

- Construye un autómata que representa el patrón de una expresión regular dada.
- Procesa la expresión regular para construir transiciones entre estados.

2. **Visualización de Autómatas**:
   - Visualiza el autómata generado como una imagen PNG, mostrando estados y transiciones.
   - Guarda la imagen en una carpeta temporal y la muestra al usuario.
3. **Validación de Expresiones Regulares**:
   - Valida la sintaxis de la expresión regular antes de generar el autómata.
   - Maneja excepciones como paréntesis desbalanceados o caracteres inválidos.
4. **Simulación de Autómatas**:
   - Simula la ejecución del autómata con una cadena de entrada.
   - Verifica si la cadena es aceptada o rechazada según las transiciones del autómata.

5. Manejo de Excepciones

El proyecto incluye manejo de excepciones para asegurar operaciones correctas, incluso con errores o entradas inválidas:

- `InvalidRegexException`: Se lanza cuando la expresión regular contiene caracteres no permitidos o paréntesis desbalanceados.
- `InvalidCharacterError`: Se lanza al encontrar un carácter inválido en una cadena de entrada durante la simulación del autómata.
- `InvalidTransitionError`: Se lanza si la cadena de entrada no termina en un estado de aceptación.

6. Pruebas

6.1 Pruebas de Funcionalidad

Se han implementado pruebas unitarias para verificar el correcto funcionamiento de las principales funciones del sistema:

- **Prueba de Generación de Autómata**: Verifica la correcta generación del autómata a partir de una expresión regular.
- **Prueba de Visualización**: Comprueba la generación y guardado correcto de la imagen del autómata.
- **Prueba de Validación de Expresiones Regulares**: Asegura que las expresiones regulares inválidas sean rechazadas con las excepciones correspondientes.
- **Prueba de Simulación de Autómatas**: Prueba la correcta aceptación o rechazo de cadenas de entrada por el autómata.

6.2 Prueba de Ejemplo

Una de las pruebas incluye la expresión regular `"(a|b)*abb"`:

- El sistema genera un autómata con dos estados y transiciones entre ellos.
- La cadena "abb" es procesada correctamente y aceptada por el autómata.

7. Conclusión

El proyecto proporciona una herramienta funcional para generar y visualizar autómatas a partir de expresiones regulares. Incluye validación de entradas y simulación de autómatas, permitiendo a los usuarios interactuar eficientemente con las expresiones regulares y comprender el comportamiento de los autómatas generados.

8. Librerías Utilizadas

El proyecto utiliza varias librerías para facilitar la construcción y visualización de autómatas, así como la manipulación de expresiones regulares. A continuación, se describen las principales:

8.1 `re` (Expresiones Regulares de Python)

La librería estándar `re` se emplea para manejar expresiones regulares. Proporciona funciones para buscar, coincidir y manipular cadenas de texto

mediante patrones. Las funciones más relevantes incluyen:

- `re.match()`: Busca una coincidencia al inicio de una cadena.
- `re.search()`: Busca una coincidencia en cualquier parte de la cadena.
- `re.findall()`: Encuentra todas las coincidencias de una expresión regular en una cadena.

Se usa para validar expresiones regulares y hacer coincidir cadenas de entrada con patrones definidos.

8.2 `graphviz` (Visualización de Gráficas)

`graphviz` se utiliza para generar y visualizar autómatas. Permite crear diagramas de grafos fácilmente, lo cual es útil para representar autómatas finitos visualmente. En el proyecto se usa para:

- Crear el gráfico de estados y transiciones del autómata.
- Renderizar el gráfico en formato PNG para guardarlo y mostrarlo al usuario.
- **Funcionalidades clave:**
  - `graphviz.Digraph`: Define el gráfico dirigido que representa los estados y transiciones del autómata.
  - `render()`: Renderiza el gráfico en el formato especificado (PNG en este caso).

Esta librería es esencial para visualizar el autómata generado a partir de la expresión regular.

8.3 `PIL` (Python Imaging Library)

`PIL` (ahora conocida como `Pillow`) se usa para manipular imágenes. Aunque principalmente se emplea para generar imágenes de autómatas, ofrece diversas funciones útiles como creación, edición y almacenamiento de imágenes. Se utiliza para:

- Guardar el gráfico generado con `graphviz` en formato PNG.
- Procesar la imagen y asegurar su correcto almacenamiento.

8.4 `os` y `tempfile` (Manejo de Archivos y Directorios)

`os` y `tempfile` se usan para gestionar archivos y directorios temporales. `os` interactúa con el sistema operativo, crea directorios y maneja rutas de archivos, mientras que `tempfile` crea archivos temporales para guardar las imágenes generadas del autómata.

- `os`: Crea directorios y maneja rutas de archivos.
- `tempfile`: Crea archivos temporales para almacenar los autómatas antes de su visualización.

8.5 `pydot` (Interfaz con Graphviz)

Aunque `graphviz` es la librería principal para generar gráficos, `pydot` se usa en algunos casos para facilitar la manipulación de gráficos de Graphviz. `pydot` simplifica la creación de grafos DOT y su conversión a otros formatos, como PNG.

- `pydot.Dot`: Define grafos de manera programática antes de pasarlos a Graphviz.

8.6 `pytest` (Pruebas Unitarias)

`pytest` se utiliza para realizar pruebas unitarias. Con esta librería, se validan las funciones y características del sistema, asegurando el correcto funcionamiento de la generación y simulación de autómatas.

- **Funciones clave**:
  - `assert`: Realiza afirmaciones sobre el comportamiento esperado del código.
  - `pytest.mark.parametrize`: Ejecuta una prueba con múltiples valores de entrada, asegurando la robustez del sistema.

Esta librería facilita el proceso de prueba y garantiza la fiabilidad del sistema.

9. Errores y Problemas Encontrados

Durante el desarrollo del proyecto, surgieron problemas relacionados con la librería `graphviz` y el manejo de expresiones regulares. Estos inconvenientes afectaron la visualización del autómata y la correcta interpretación de ciertos patrones. A continuación, se detallan los errores encontrados:

9.1 Camino Doble en `graphviz`

Un problema significativo ocurrió con la librería `graphviz`, específicamente en la generación de los caminos del autómata. Algunas transiciones entre estados generaban un camino doble (dos flechas que iban del mismo estado a otro con el mismo símbolo de entrada). Este comportamiento no deseado afectaba la claridad de la visualización.

Posible Causa:
Este error parece estar relacionado con la forma en que `graphviz` maneja la creación de los diagramas de estado y transiciones. A pesar de intentar varias soluciones, como forzar la exclusividad de las transiciones o modificar las propiedades de los nodos, el error persistió y no se pudo corregir completamente.

Impacto en el Proyecto:
El camino doble afectaba la estética y claridad de los autómatas generados. Sin embargo, no comprometió la funcionalidad del proyecto, ya que las transiciones seguían siendo correctas a nivel lógico.

9.2 Problemas con Expresiones Regulares que Inician con `*`

Otro problema fue la incapacidad de manejar correctamente expresiones regulares que iniciaran con el carácter `*`. Al usar `*` al inicio de la expresión regular (que debería permitir que un patrón coincida con cero o más ocurrencias del mismo símbolo), el autómata siempre comenzaba desde el estado 1 y luego se dirigía hacia el estado 0, lo cual no reflejaba el comportamiento esperado de "cero a muchas" ocurrencias.

Descripción del Problema:

- Cuando una expresión regular comenzaba con `*`, el autómata generaba un estado inicial en el estado 1, seguido de transiciones de 1 a 0 a muchos. Esto no representaba correctamente un autómata con transición de cero a muchas ocurrencias. La representación correcta debería haber comenzado en el estado 0.
- Esta anomalía se debía a una interpretación incorrecta de la expresión regular al crear el autómata. El `*` se aplicaba de manera equivocada al primer símbolo, iniciando el autómata con un estado no deseado.

Impacto en el Proyecto:
Este error afectaba a las expresiones regulares que empleaban el carácter `*` al principio, ya que no representaba con precisión el comportamiento de un autómata con transiciones desde el estado inicial a cero o más ocurrencias del símbolo.

Solución Intentada:
Se probaron diferentes enfoques para corregir este comportamiento, como modificar la generación de transiciones iniciales y ajustar el manejo del carácter `*` en el código. Sin embargo, debido a limitaciones en la forma en que `graphviz` maneja los estados, el problema persistió en algunos casos.