



unitec[®]
LAUREATE INTERNATIONAL UNIVERSITIES[®]

Jesús Orlando Cueva Villela

21741211

Proyecto de clase: Evaluador de autómatas

31 de agosto de 2020

Ing. Cesar Orellana

Teoría de la Computación

Índice

Introducción	3
¿En qué puedo usar un autómata?	3
Tecnologías usadas	4
Archivo start.py.....	6
Épsilon NFA.....	8
NFA.....	10
DFA.....	13
Pruebas.....	15
Prueba #1	15
Prueba #2	16
Prueba #3.....	17
Prueba #4	18
Prueba #5	19
Tiempos de ejecución	20
Comparación.....	20
Anexos.....	21

Introducción

A continuación, se presenta el desarrollo sobre el proyecto de Equivalencia entre autómatas DFA, NFA-E, y Expresiones Regulares. Este se llevó a cabo como proyecto para la clase de teoría de la computación. Este proyecto tiene como objetivo implementar algoritmos para encontrar equivalencia en los lenguajes generados por autómatas DFA, NFA-E, y Expresiones Regulares. También se detalla sobre el desarrollo de este, tomando en cuenta las tecnologías usadas para su realización, las pruebas, el entorno de desarrollo.

La teoría de autómatas es una rama de las ciencias de la computación que estudia las máquinas abstractas y los problemas que estas son capaces de resolver. La teoría de autómatas está estrechamente relacionada con la teoría del lenguaje formal ya que los autómatas son clasificados a menudo por la clase de lenguajes formales que son capaces de reconocer.

Un autómata es un modelo matemático para una máquina de estado finito (FSM por sus siglas en inglés). Una FSM es una máquina que, dada una entrada de símbolos, se mueve a través de una función de transición. Esta función de transición dice al autómata a qué estado cambiar dados unos determinados estados y símbolos.

Dependiendo del estado en el que el autómata finaliza se dice que este ha aceptado o rechazado la entrada. Si éste termina en el estado de “aceptación”, el autómata acepta la palabra. Si lo hace en el estado de “rechazo”, el autómata rechazó la palabra, el conjunto de todas las palabras aceptadas por el autómata constituyen el lenguaje aceptado por el mismo.

¿En qué puedo usar un autómata?

Su creador Alan Turing padre de la computación, estudió una máquina abstracta que poseía la misma capacidad de las computadoras actuales. Su objetivo era determinar la frontera entre lo que puede y no puede hacer una computadora, y aun cuando sus estudios están basados en estas máquinas abstractas son aplicables hoy en día a nuestros computadores.

Es una de las bases de la inteligencia artificial, los autómatas suponen un estudio más avanzado para la creación de la misma, otro gran ejemplo es en la industria, en cualquier sistema de gestión tecnológica puesto que poseen un grado de inteligencia para hacer las cosas, en centros de investigación para la búsqueda de patrones, la cual fue una de las principales causas de su creación ya que Turing previó la pérdida de tiempo al usar recurso humano para hacer búsquedas exhaustivas y fue así que surgió la creación de los autómatas. Pero si aún no queda claro, un claro ejemplo de un autómata es cuando interactúas con una computadora, en cualquier juego tu contra la computadora, más específico en un juego de ajedrez puesto que es en si tu contra un autómata programado.

Tecnologías usadas

Lenguaje de programación Python

Python es un lenguaje de programación de código abierto, orientado a objetos, muy simple y fácil de entender. Tiene una sintaxis sencilla que cuenta con una vasta biblioteca de herramientas, que hacen de Python un lenguaje de programación único.

Una de las ventajas principales de aprender Python es la posibilidad de crear un código con gran legibilidad, que ahorra tiempo y recursos, lo que facilita su comprensión e implementación.

JSON

Formato de texto sencillo para el intercambio de datos. Gracias a esta implementación, resulta mucho más sencillo escribir un analizador sintáctico para él. Al ser JSON un formato muy extendido para el intercambio de datos, se han desarrollado Api para distintos lenguajes, en nuestro caso Python, que permiten analizar sintácticamente, generar, transformar y procesar este tipo de dato.

Tkinter

Es un binding de la biblioteca gráfica Tcl/Tk para el lenguaje de programación Python. Se considera un estándar para la interfaz gráfica de usuario (GUI) para Python y es el que viene por defecto con la instalación para Microsoft Windows.

GitHub

GitHub es un servicio de alojamiento de repositorios de Git, pero agrega muchas de sus propias características. Si bien Git es una herramienta de línea de comandos, GitHub proporciona una interfaz gráfica basada en web. También proporciona control de acceso y varias funciones de colaboración, como wikis y herramientas básicas de gestión de tareas para cada proyecto.

Networkx

Es una biblioteca de Python para el estudio de gráficos y análisis de redes. Networkx es un software libre. Es adecuado para operar grandes gráficos del mundo real. Debido a su dependencia en una estructura de datos un “diccionario de diccionario” de Python puro, Networkx es un marco razonablemente eficiente, muy escalable, muy portátil para el análisis de redes sociales y de redes.

Matplotlib

Es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

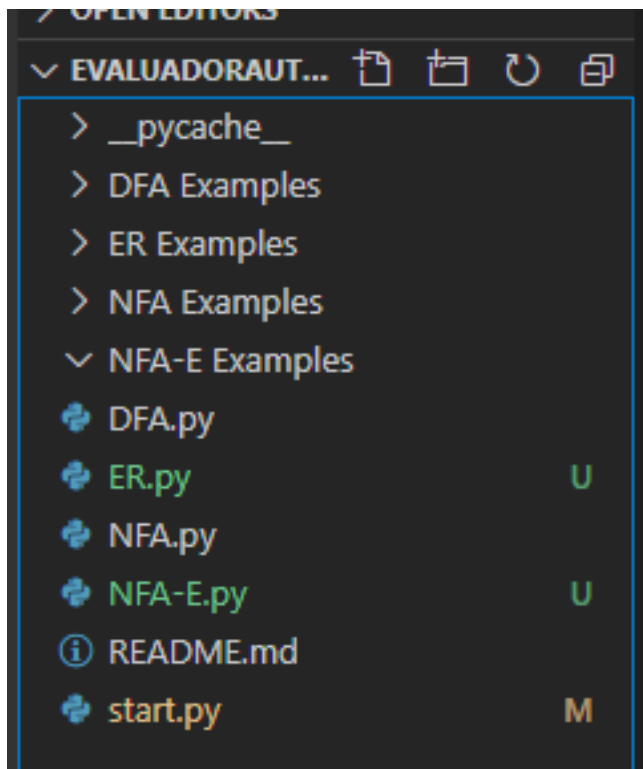
Time

Utilizamos la función `time()` del módulo estándar que lleva el mismo nombre (módulo `time`), esta función nos retorna un valor de tipo flotante con el tiempo que lleva ejecutándose nuestra máquina.

Visual Studio Code

Es un editor de código fuente desarrollado por Microsoft para Windows, Linux y macOS. Incluye soporte para la depuración, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código.

Código



El proyecto consta de 5 archivos necesarios para su funcionamiento. La estructura del proyecto está dada por la figura 1, en la cual se observan 5 archivos .py (Python). Adicionalmente, el proyecto incluye carpetas nombradas DFA Examples, NFA Examples, NFA-E Examples, y ER Examples, los cuales contienen archivos .json que son utilizados para las pruebas del proyecto.

Figura 1. Fuente: Propia

Archivo start.py

```
start.py x
start.py > main
1 # IMPORTS #
2 from DFA import *
3 from NFA import *
4 import json
5 import tkinter as tk
6 from tkinter import filedialog
7
8 def main():
9     print("Bienvenido")
10    print("Porfavor ingresa tu archivo: ")
11
12    root = tk.Tk()
13    root.withdraw()
14    selectedFile = filedialog.askopenfilename()
15    #Opening JSON File
16    f = open(selectedFile,)
17
18    #Returns JSON object as a dictionary
19    data = json.load(f)
20
21    alphabet = data["alphabet"]
22    states = data["states"]
23    initial_states = data["initial_state"]
24    final_states = data["accepting_states"]
25    transitions = data["transitions"]
26
27    print("Ingresa la cadena que quieres evaluar: ")
28    str_test = input()
29
30    print("\nEvaluando automata...")
31
```

El archivo start.py se encarga de llamar a todas las clases externas necesarias para hacer uso de sus funciones. Al empezar el documento vemos los imports necesarios para ejecutar el programa.

También hacemos uso de la librería json, la cual es necesaria para hacer lectura a los archivos .json de donde leemos la información de los autómatas. La librería tkinter es utilizada por el beneficio que brinda al poder crear ventanas visuales al usuario. En esta ventana, el usuario puede buscar intuitivamente el archivo deseado.

Figura 2. Fuente: Propia

Luego definimos la función main, la cual ejerce la función de llamar a todos los métodos necesarios para la ejecución del programa. Inicialmente, se imprime un mensaje de bienvenida, con finalidades de ser fácil de usar. Las líneas 12, 13, 14 realizan la acción de mostrar una ventana interactiva al usuario donde este pueda seleccionar el archivo que desea evaluar.

Seguidamente se le muestra al usuario un mensaje que ingrese el archivo JSON que desea evaluar. Recordemos que en este archivo se encuentra definido un autómata.

```
1  {
2    "alphabet": [
3      "0",
4      "1"
5    ],
6    "states": [
7      "q0",
8      "q1"
9    ],
10   "initial_state": "q0",
11   "accepting_states": [
12     "q1"
13   ],
14   "transitions": [
15     ["q0", "1", "q1"],
16     ["q1", "0", "q1"],
17     ["q1", "1", "q1"]
18   ]
19 }
```

En la figura 3 podemos observar un ejemplo de un archivo JSON utilizado. El autómata presentado es un DFA utilizado en las pruebas del proyecto. El archivo JSON consta de un objeto que contiene un arreglo de Strings. Entre ellos, definimos nuestro alfabeto, el varía de autómata a autómata, los estados del autómata, el estado inicial y final del autómata, y por ultimo las transiciones del autómata. Cabe aclarar, que estas 5 son necesarias para nuestro autómata, pero los valores que contienen pueden variar.

Figura 3. Fuente: Propia

Luego, el bloque de código entre las líneas 21-25 consta de la declaración de variables donde almacenamos la información proveniente del JSON para su manipulación. Pedimos al usuario ingresar una cadena que desea evaluar en nuestro autómata resultante. Luego hacemos llamada a las funciones necesarias. Dado que sea un autómata NFA-E (que contenga el símbolo ' ϵ ') hacemos llamada a esta clase, o dado el caso sea un NFA hacemos llamada a esta clase. Por último, en el archivo, hacemos llamada a la propia función main () para que se ejecute.

Épsilon NFA

```
def enfa2nfa(self, alphabet, states, initial_state, accepting_states, transitions, str_test):...
```

Figura 4. Fuente: Propia

En el archivo ENFA.py definimos nuestra clase para la resolución del autómata Épsilon NFA. Esta función recibe de parámetros, el parámetro self, requerido por Python, luego el alfabeto del autómata, los estados, los estados iniciales, los estados finales, las transiciones, y por ultimo la cadena a evaluar. Es importante mencionar que esta cadena no se utiliza hasta que nuestro autómata está en la forma DFA y ahí evaluamos nuestra cadena. Entonces este parámetro solo se mueve a través de las funciones hasta llegar al punto que lo queramos evaluar.

Figura 5. Fuente: Propia

```
start.py  ENFA.py  NFA.py
ENFA.py > ENFA
1  import copy
2  from NFA import *
3  from time import time
4  import networkx as nx
5  import matplotlib.pyplot as plt
6
7  class ENFA:
```

Al principio del archivo tenemos las importaciones de librerías necesarias para el proyecto. La librería time es utilizada para la medición de tiempo de ejecución de la función. La librería networkx y matplotlib son utilizadas para graficar el resultado. En este caso, hacemos uso de la

librería copy para realizar una copia de las variables que nos vienen. Es importante recordar que estos parámetros son los valores tomados del JSON, dicho de otra forma, es nuestro autómata. La razón de hacer esta copia es por que al trabajar con los parámetros las variables se cambian de forma que nuestro autómata también cambia, cambiándonos todo nuestro autómata y resultando en un autómata completamente diferente. Entonces, de la línea 11-15 hacemos esta copia y lo guardamos en una variable nueva la cual podemos manipular. Dado que este es un autómata épsilon, y lo queremos convertir a un NFA, en la línea 18, leemos nuestro alfabeto y removemos el símbolo “E”, el cual en nuestro caso se denomina “Épsilon”. En la línea 21 le damos clear a la tabla de transiciones, dado que estaremos creando transiciones nuevas a lo largo del proceso.

```
def enfa2nfa(self, alphabet, states, initial_state, accepting_states, transitions, str_test):

    tiempo_inicial = time()
    print(tiempo_inicial)

    #Hacemos la copia del archivo para manipular los datos
    alpha = copy.deepcopy(alphabet)
    stat = copy.deepcopy(states)
    i_stat = copy.deepcopy(initial_state)
    a_stat = copy.deepcopy(accepting_states)
    trans = copy.deepcopy(transitions)

    #Borrar epsilon del alfabeto listo
    alphabet.remove("E")

    #Borrar transiciones con epsilon
    transitions.clear()
    for t in trans:
        if t[1] != "E":
            transitions.append(t)

    G = nx.MultiDiGraph()
```

Figura 6. Fuente: Propia

Seguidamente, ahora guardamos todas aquellas transiciones que no tenga Épsilon.

```
for a in alphabet:
    for s in stat:
        for t in trans:
            #Encontrar las transiciones con epsilon
            if s == t[0] and t[1] == "E":
                #Adonde voy con epsilon desde el estado actual
                for t2 in trans:
                    if t[2] == t2[0] and t2[1] == a:
                        #Encontramos el camino del estado con el input
                        for t3 in trans:
                            if t2[2] == t3[0] and t3[1] == "E":
                                #Por ultimo encontramos el camino donde podemos llegar con epsilon con el input del estado
                                lista = [s, a, t3[2]]
                                if lista not in transitions:
                                    transitions.append(lista)

# print("E-NFA -> NFA")
# print("Alfabeto: ", alphabet)
# print("Estados: ", states)
# print("Estados iniciales: ", initial_state)
# print("Estados finales: ", accepting_states)
# print("Transiciones: ", transitions)

nfa = NFA()
nfa.nfa2dfa(alphabet, states, initial_state, accepting_states, transitions, str_test)
```

Figura 7. Fuente: Propia

En este bloque de código realizamos el procedimiento de equivalencia E-NFA -> NFA. Para empezar, recorreremos nuestro alfabeto para conocer los "inputs" que tengamos. En el

segundo for recorreremos nuestros estados y en el tercer for recorreremos nuestras transiciones. Entonces, nos fijamos en el estado con su transición, en este caso con épsilon. ¿Similar a decir "Con el estado X adonde puedo ir con Épsilon?". Si se encuentra un camino, realizamos un nuevo for para recorrer estas transiciones. Nuevamente nos preguntamos, "¿Adonde puedo ir con este estado y con mi input actual?" Recordemos que el input es cada valor en nuestro alfabeto. Si se encuentra un camino, guardamos estos valores en formato, [estado, alfabeto, llegada]. Si estas nuevas combinaciones no están en nuestro diccionario de transiciones, las agregamos, dado que esto es una nueva transición en nuestro NFA. El bloque de código comentado es para observar la nueva información.

NFA

```
nfa = NFA()
nfa.nfa2dfa(alphabet, states, initial_state, accepting_states, transitions, str_test)
```

Figura 8. Fuente: Propia

La declaración de la función NFA se realiza de la siguiente manera. Recibe el alfabeto, estados, estados iniciales, estados finales, transiciones, y cadena a evaluar. Es importante mencionar que, a este punto, nuestro autómata proviene de la resolución de una épsilon nfa, por lo tanto, nuestro diccionario es un poco diferente al ingresado inicialmente. También, que nuestra variable str_test no ha sufrido ningún cambio. Figura 9. Fuente: Propia

```
start.py  ENFA.py  NFA.py  X
NFA.py > ...
1  import copy
2  from DFA import *
3  import networkx as nx
4  import matplotlib.pyplot as plt
5  from time import time
6
```

En nuestro archivo hacemos uso de las librerías copy, networkx, matplotlib, y time. También hacemos llamada a la clase DFA para su posterior uso. La librería copy se utiliza para hacer una copia de los parámetros que nos vienen. Como se había

comentado anteriormente, se hace esta copia para evitar algún conflicto en el posterior uso de las variables. Las copias se guardan en una variable nueva con el fin de poder manipularlas. La librería networkx y matplotlib son utilizadas para graficar el resultado del autómata. Por último, la librería time se utiliza para llevar el control de tiempo de ejecución de la función.

```
class NFA:
    def __init__(self):
        pass

    def union(self, alfabeto, estados, transiciones):
        res = []

        for alfa in alfabeto:
            tr = []
            for t in transiciones:
                if type(estados) != type(res):
                    if t[0] == estados and t[1] == alfa:
                        tr.append(t[2])
                        #Hay una transicion entre Estado y Alfabeto
                        #Entonces guardamos adonde nos lleva
                    else:
                        for index in estados:
                            if t[0] == index and t[1] == alfa:
                                tr.append(t[2])
                                #Hay una transicion entre Estado y Alfabeto
                                #Entonces guardamos adonde nos lleva
            if len(tr) == 0:
                tr.append("/")
                #Si no hay transicion guardamos un valor nulo
            res.append(tr)
        var = [estados, res]

        return var
        #Retornamos el estado con sus transacciones
```

Figura 10. Fuente: Propia

Se hace instancia de la clase NFA, y vemos la función unión, la cual recibe de parámetro, el parámetro inicial propio de Python, el alfabeto de nuestro autómata, los estados y transiciones.

El fin de esta función es realizar la unión de los estados que se puedan crear al momento de realizar la equivalencia entre el NFA y el DFA.

Creamos una lista vacía en la cual estaremos guardando el resultado. Recorremos nuestro alfabeto y creamos una lista auxiliar. Recorremos nuestras transiciones y nos fijamos en que nuestro estado tenga una transición existente con nuestro input (el carácter del alfabeto actual), si existe dicha transición, la guardamos en la lista. Si no hay ninguna transición, dado que esto proviene de un NFA-E, en nuestra lista guardamos esta transición con un carácter que nosotros sabemos que representa una transición nula. Dicho carácter en nuestro ejemplo es '/'. Guardamos estas nuevas transiciones en nuestra lista resultado. Las guardamos con el formato, 1: Estado 2: Valor 3: Destino, en otras palabras, estados concatenados con las nuevas transiciones creadas. Al finalizar retornamos esta lista.

Figura 11. Fuente: Propia

```
def list2String(self, s):  
    str1 = ""  
    for ele in s:  
        str1 += ele  
    return str1
```

La función list2String realiza la acción de convertir una lista de Python y la convierte a un string. Esto es para el momento de graficar, se facilite la acción ya que matplotlib no puede graficar listas como tal, se necesitaba un tipo de valor valido, tal como string.

Figura 12. Fuente: Propia

```
def nfa2dfa(self, alphabet, states, initial_state, accepting_states, transitions, str_test):  
  
    tiempo_inicial = time()  
    print(tiempo_inicial)  
  
    #Hacemos la copia del archivo para manipular los datos  
    alpha = copy.deepcopy(alphabet)  
    stat = copy.deepcopy(states)  
    i_stat = copy.deepcopy(initial_state)  
    a_stat = copy.deepcopy(accepting_states)  
    trans = copy.deepcopy(transitions)  
  
    #Borramos datos para pasar NFA -> DFA  
    states.clear()  
    accepting_states.clear()  
    transitions.clear()  
  
    var = self.union(alpha, i_stat, trans)  
    nfaTable = [var]  
  
    # nfaTable = [x = estado] [y = transiciones de x]  
  
    for x in nfaTable:      #estados  
        for y in x[1]:  
            existe = True  
            for z in nfaTable:  
                if y == z[0]:      #transiciones  
                    existe = False #revisamos si existe el nuevo estado
```

La función nfa2dfa realiza la equivalencia entre un autómata nfa y un dfa. Las primeras líneas que encontramos son las de las variables del tiempo. Utilizadas para medir el tiempo de ejecución de la función. Luego hacemos copia de las variables y poder

transformarlas en variables manipulables. Borramos las transiciones existentes, ya que se estarán creando transiciones nuevas. Guardamos en una variable la lista resultante de nuestra función unión. Con esta nueva data creamos una nueva lista, 'nfaTable' la cual almacenara nuestros nuevos datos. 'nfaTable' es una lista bidimensional en la cual estaremos guardando los estados, y las transiciones con ese estado.

Figura 13. Fuente: Propia

```
for x in nfaTable: #estados
    for y in x[1]:
        existe = True
        for z in nfaTable:
            if y == z[0]: #transiciones
                existe = False #revisamos si existe el nuevo estado

        if existe:
            res = self.union(alpha, y, trans)
            if res not in nfaTable:
                nfaTable.append(res)

#Eliminamos alguna transición repetida
newtable=[]
for index in range(len(nfaTable)):
    for ind in range(index, len(nfaTable)):
        if nfaTable[index] == nfaTable[ind]:
            newtable.append(nfaTable[ind])
            break
```

Recorremos nuestra lista `nfaTable` y en ella realizamos la siguiente condición: si hay algún estado en esta posición utilizamos una variable flag, 'existe' y continuamos, utilizamos un nuevo for para recorrer la lista otra vez, pero ahora en busca de las transiciones. Si el estado en el que estamos tiene alguna transición entonces existe es seteada a falsa.

Dado el caso que existe sea verdadero, se procede a crear la nueva transición y estado y se agrega a nuestra lista. Durante el desarrollo del programa se encontró el problema de que se estaba guardando dos veces la información. Para la resolución de este problema se realizó el siguiente bloque de código, el cual crea una nueva lista, y recorre nuestra lista `nfaTable`. El segundo ciclo es para realizar la localización del índice actual en la lista de `nfaTable`. Entonces lo que se realiza es que si el índice en las listas coincide agregaremos el valor a nuestra nueva lista.

Figura 14. Fuente: Propia

```
#Creamos los nuevos estados, con sus transiciones
for x in nfaTable:

    if x[0] not in states:
        states.append(x[0])

    for i in range(len(alpha)):
        tran=[x[0],alpha[i],x[1][i]]
        if x[1][i] not in states:
            states.append(x[1][i])

        if tran[2] != "/":
            transitions.append(tran)

    for x in states:
        for y in a_stat:
            if y in x:
                accepting_states.append(x)

G = nx.MultiDiGraph()

for x in states:
    n = self.convert(x)
    G.add_node(n)
```

El siguiente bloque de estado es para crear los nuevos estados con sus transiciones. A este punto, ya es un autómata DFA. Recorremos nuestra tabla `nfaTable` y realizamos lo siguiente:

Si el estado actual no está en los estados, lo agregamos a nuestra lista de estados del autómata. El siguiente ciclo es para las transiciones.

Guardamos en una variable, el estado, con su input actual, ósea el valor actual del alfabeto, y el valor de destino. Nuevamente, verificamos que si este valor de destino no está en los estados del autómata, lo

agregamos. Luego, agregamos todo valor destino siempre y cuando sea destino a '/', dado que este carácter significa una transición nula. Los estados de aceptación se guardan en el autómata y este valor se envía al DFA. Las siguientes líneas son las instrucciones de la librería para poder graficar el resultado. Las últimas líneas son para la variable de tiempo y el cálculo del tiempo de ejecución de la función.

DFA

```
dfa = DFA()
dfa.dfa_evaluate(alphabet, states, initial_state, accepting_states, transitions, str_test)
```

Figura 15. Fuente: Propia

El ultimo paso de nuestra solución es evaluar el autómata. Para eso hacemos uso de la función, `dfa_evaluate`. La cual recibe de parámetros, el alfabeto, estados, estados iniciales, estados finales, transiciones, y la cadena a evaluar. Cabe mencionar que estos son provenientes de la función `nfa2dfa`, y la cadena a evaluar es utilizada en esta función.

Figura 16. Fuente: Propia

```
DFA.py  X
DFA.py > ...
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 from time import time
```

En el archivo DFA se importan las siguientes librerías: `networkx`, `matplotlib`, y `time`.

Las cuales son utilizadas para graficar el resultado y medir el tiempo de ejecución,

respectivamente.

Figura 17. Fuente: Propia

```
class DFA:
    def __init__(self):
        pass

    def list2String(self, s):
        str1 = ""
        for ele in s:
            str1 += ele
        return str1
```

La función `list2String` realiza la acción de convertir una lista de Python y la convierte a un string. Esto es para el momento de graficar, se facilite la acción ya que `matplotlib` no puede graficar listas como tal, se necesitaba un tipo de valor valido, tal como string.

Figura 18. Fuente: Propia

```
def dfa_evaluate(self, alphabet, states, initial_state, accepting_states, transitions, str_test):

    tiempo_inicial = time()
    print(tiempo_inicial)

    current_state = initial_state
    G = nx.MultiDiGraph()

    for x in states:
        n = self.convert(x)
        G.add_node(n)

    for char_index in range(len(str_test)):
        current_char = str_test[char_index]
        for t in transitions:
            next_state = t[2]
            current_state = next_state

    if current_state in accepting_states:
        print ("Pertenece a L(M)")
    else:
        print ("No pertenece a L(M)")
```

Iniciamos declarando las variables del tiempo.

Configuramos nuestra variable de estado actual con la de estado inicial, ya que es el estado del cual partimos. Luego declaramos nuestro gráfico. Agregamos los nodos de los estados y transiciones a nuestro gráfico. Luego la lógica de la evaluación del autómata. El ciclo se realiza de acuerdo con la longitud de la cadena a evaluar. Configuramos nuestro carácter actual como la posición actual dentro de la cadena. Hacemos un ciclo para recorrer nuestras transiciones, ya que estas nos dirán los caminos que podemos tomar en nuestro autómata. Configuramos que nuestro estado siguiente es el de la posición 3, 't[2]' en nuestra lista, ya que es el estado destino. Y nuestro estado actual pase a ser el estado siguiente.

Luego evaluamos estas condiciones, si nuestro estado actual se encuentra en un estado de aceptación, podemos decir que nuestro autómata pertenece al lenguaje, caso contrario no pertenece a nuestro lenguaje.

Figura 19. Fuente: Propia

```
tiempo_final = time()
print(tiempo_final)
tiempo_ejecucion = tiempo_final - tiempo_inicial
print("El tiempo de ejecucion de la funcion DFA fue: ", tiempo_ejecucion)

nx.draw(G, with_labels=True)
plt.ion()
plt.show()
plt.pause(0.001)
input("Press [enter] to continue.")
#plt.savefig("DFA.png", format="PNG")
|
pass
```

Por último, tenemos las variables de tiempo y las sentencias para graficar el resultado. The plt.ion() refers to turn the interactive mode on. plt.show() muestra el grafico en pantalla.

Pruebas

```
SimpleENFA_1.json X
NFA-E Examples > SimpleENFA_1.json > [
1  {
2    "alphabet": [
3      "a",
4      "b",
5      "q",
6      "E"
7    ],
8    "states": [
9      "p",
10     "q",
11     "r",
12     "s",
13     "t",
14     "u",
15     "v",
16     "w",
17     "x",
18     "y"
19   ],
20   "initial_state": "p",
21   "accepting_states": [
22     "y"
23   ],
24   "transitions": [
25     ["p", "E", "q"],
26     ["p", "E", "r"],
27     ["q", "E", "s"],
28     ["r", "b", "t"],
29     ["s", "q", "u"],
30     ["t", "a", "v"],
31     ["u", "E", "w"],
32     ["v", "E", "x"],
33     ["w", "E", "y"],
34     ["x", "E", "y"],
35     ["p", "E", "p"],
36     ["q", "E", "q"],
37     ["r", "E", "r"],
38     ["s", "E", "s"],
39     ["t", "E", "t"],
40     ["u", "E", "u"],
41     ["v", "E", "v"],
42     ["w", "E", "w"],
43     ["x", "E", "x"],
44     ["y", "E", "y"]
45   ]
46 }
```

Prueba #1

Dado el archivo JSON:

Tiempo de ejecución 1:

1.7145156860351562

Prueba 1: ba

Tiempo de ejecución 2:

2.0561485290527344

Prueba 2: q

Tiempo de ejecución 3:

1.5779283046722412

Prueba 3: baq

Tiempo de ejecución 4:

1.8760523796081543

Prueba 4: baaaaaaa

Tiempo de ejecución 5:

1.6457233428955078

Prueba 5: qqqqqqqqqqqq

```
start.py SimpleENFA_2.json X
NFA-E Examples > SimpleENFA_2.json > [
1  {
2      "alphabet": [
3          "a",
4          "b",
5          "E"
6      ],
7      "states": [
8          "s0",
9          "s1",
10         "s2",
11         "s3",
12         "s4",
13         "s5",
14         "s6",
15         "s7"
16     ],
17     "initial_state": "s0",
18     "accepting_states": [
19         "s7"
20     ],
    "transitions": [
        ["s0", "E", "s1"],
        ["s0", "E", "s4"],
        ["s1", "a", "s2"],
        ["s2", "a", "s2"],
        ["s2", "b", "s2"],
        ["s2", "a", "s3"],
        ["s3", "E", "s7"],
        ["s4", "b", "s5"],
        ["s5", "a", "s5"],
        ["s5", "b", "s5"],
        ["s5", "b", "s6"],
        ["s6", "E", "s7"],
        ["s0", "E", "s0"],
        ["s1", "E", "s1"],
        ["s2", "E", "s2"],
        ["s3", "E", "s3"],
        ["s4", "E", "s4"],
        ["s5", "E", "s5"],
        ["s6", "E", "s6"],
        ["s7", "E", "s7"]
    ]
}
```

Tiempo de ejecución 1:
5.376879453659058

Tiempo de ejecución 2: 6.75682806968689

Tiempo de ejecución 3:
3.483631134033203

Tiempo de ejecución 4:
4.039930105209351

Tiempo de ejecución 5:
3.167973041534424

Prueba 5:
abbbbbbaaaaaaaaaabbbbbbbbaaaaaa


```

start.py SimpleENFA_3.json X
NFA-E Examples > {} SimpleENFA_3.json > [ ] tr
1  {
2    "alphabet": [
3      "1",
4      "0",
5      "E"
6    ],
7    "states": [
8      "q0",
9      "q1",
10     "q2",
11     "q3",
12     "q4"
13   ],
14   "initial_state": "q0",
15   "accepting_states": [
16     "q2"
17   ],
18   "transitions": [
19     ["q0", "E", "q2"],
20     ["q0", "1", "q1"],
21     ["q1", "1", "q0"],
22     ["q2", "0", "q3"],
23     ["q2", "1", "q4"],
24     ["q3", "0", "q2"],
25     ["q4", "0", "q2"],
26     ["q0", "E", "q0"],
27     ["q1", "E", "q1"],
28     ["q2", "E", "q2"],
29     ["q3", "E", "q3"],
30     ["q4", "E", "q4"]
31   ]
32 }

```

Prueba #3

Dado el archivo JSON:

Tiempo de ejecución 1:

4.091848373413086

Prueba 1: 1010

Tiempo de ejecución 2:

2.8230323791503906

Prueba 2: 01010101010101

Tiempo de ejecución 3:

2.045043706893921

Prueba 3: 1111111110000000001111111

Tiempo de ejecución 4:

2.1211555004119873

Prueba 4: 00000011111111110000000

Tiempo de ejecución 5:

2.19345760345459

Prueba 5: 01011010

Prueba #4

```
start.py SimpleENFA_4.json X
NFA-E Examples > {} SimpleENFA_4.json > [ ]
1  {
2    "alphabet": [
3      "0",
4      "1",
5      "E"
6    ],
7    "states": [
8      "q0",
9      "q1",
10     "q2",
11     "q3",
12     "q4"
13   ],
14   "initial_state": "q0",
15   "accepting_states": [
16     "q4"
17   ],
18   "transitions": [
19     ["q0", "E", "q1"],
20     ["q0", "E", "q2"],
21     ["q1", "0", "q3"],
22     ["q2", "1", "q3"],
23     ["q3", "1", "q4"],
24     ["q0", "E", "q0"],
25     ["q1", "E", "q1"],
26     ["q2", "E", "q2"],
27     ["q3", "E", "q3"],
28     ["q4", "E", "q4"]
29   ]
30 }
```

Dado el archivo JSON:

Tiempo de ejecución 1:
3.2100348472595215

Prueba 1: 1010

Tiempo de ejecución 2:
1.991725206375122

Prueba 2: 01010101010101

Tiempo de ejecución 3:
2.175814151763916

Prueba 3:
111111111000000000111111

Tiempo de ejecución 4:
1.966780424118042

Prueba 4: 0000001111111110000000

Tiempo de ejecución 5:
2.1860368251800537

Prueba 5: 01011010

Prueba #5

```
start.py  SimpleENFA_5.json X
NFA-E Examples > SimpleENFA_5.json > ..
1  {
2      "alphabet": [
3          "a",
4          "b",
5          "E"
6      ],
7      "states": [
8          "1",
9          "2",
10         "3",
11         "4",
12         "5"
13     ],
14     "initial_state": "1",
15     "accepting_states": [
16         "5"
17     ],
18     "transitions": [
19         ["1", "E", "2"],
20         ["1", "a", "3"],
21         ["2", "a", "5"],
22         ["2", "a", "4"],
23         ["3", "b", "4"],
24         ["4", "a", "5"],
25         ["4", "b", "5"],
26         ["1", "E", "1"],
27         ["2", "E", "2"],
28         ["3", "E", "3"],
29         ["4", "E", "4"],
30         ["5", "E", "5"]
31     ]
32 }
```

Dado el archivo JSON:

Tiempo de ejecución 1:
5.1614439487457275

Prueba 1: ab

Tiempo de ejecución 2:
2.9881808757781982

Prueba 2: abba

Tiempo de ejecución 3:
2.1911957263946533

Prueba 3: aaaaaaaaaabbbbbbbbbb

Tiempo de ejecución 4:
2.111799478530884

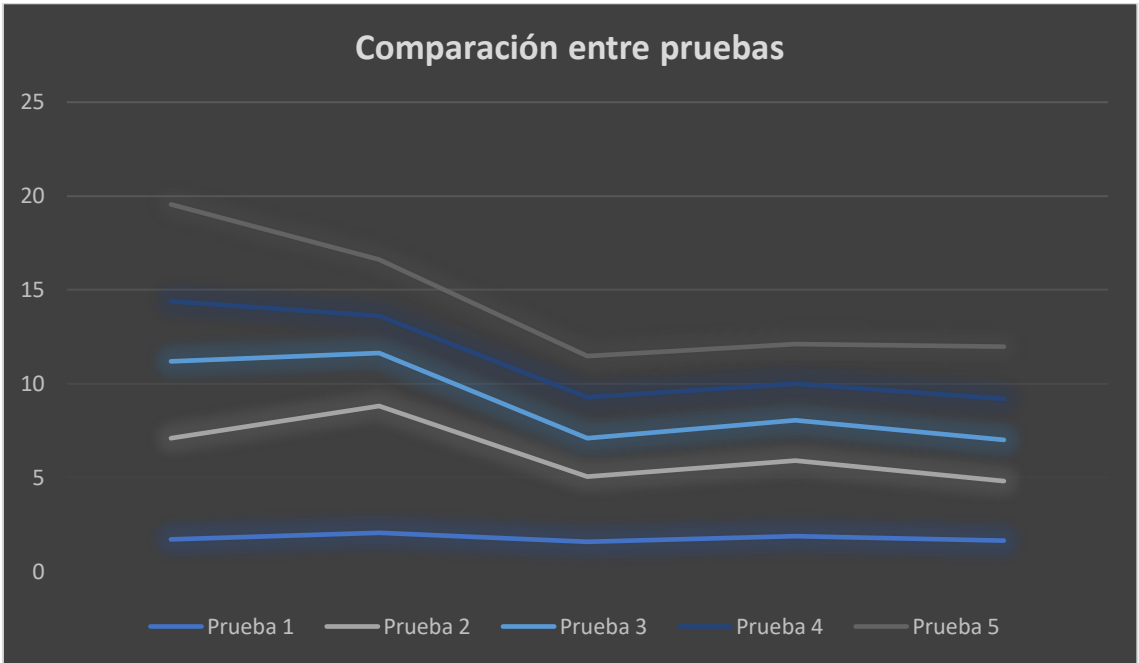
Prueba 4: bbbbbbbbbbbbbbbbbb

Tiempo de ejecución 5:
2.77309250831604

















Prueba 5: babababababaaabb

Tiempos de ejecución

Comparación



Anexos

-  <http://www.pythondiario.com/2015/06/afd-en-python-automata-finito.html>
-  <https://stackoverflow.com/questions/9319317/quick-and-easy-file-dialog-in-python>
-  <https://www.geeksforgeeks.org/read-json-file-using-python>
-  https://www.w3schools.com/python/python_conditions.asp
-  <https://www.mclibre.org/consultar/python/lecciones/python-entrada-teclado.html>
-  <https://tutorial.recursospython.com/clases/>
-  <https://www.akademus.es/blog/programacion/principales-usos-python/#:~:text=Python%20es%20un%20lenguaje%20de,un%20lenguaje%20de%20programaci%C3%B3n%20%C3%9Anico.>
-  <https://cheatography.com/mutanclan/cheat-sheets/python-regular-expression-regex/>
-  <https://es.wikipedia.org/wiki/JSON>
-  <http://www.w3big.com/es/regexp/regexp-operator.html>
-  <https://pysimpleautomata.readthedocs.io/en/latest/tutorial.html>
-  <http://delta.cs.cinvestav.mx/~mcintosh/comun/summer2006/algebraPablo.html/node6.html>
-  <http://decsai.ugr.es/~rosa/tutormc/teoria/lenguajesregulares2.html>
-  <https://jsoneditoronline.org/#right=local.honoga>
-  <https://stackoverflow.com/questions/35272592/how-are-finite-automata-implemented-in-code/35279645>
-  <https://networkx.github.io/documentation/stable/tutorial.html>