

A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

31-8-2020

# Documentación

Proyecto Automatas

Several thin, curved lines in dark blue and light grey that originate from the bottom left and sweep upwards and to the right.

Javier Nuñez  
UNITEC

# Índice

## Contenido

Índice .....	1
Introducción .....	2
1. Explicación de API y funciones Internas .....	3
1.1 Diagrama de Clases .....	3
1.2 Clase DFA .....	4
1.3 Clase NFA-e .....	7
1.4 Clase REGEX (Expresiones Regulares) .....	11
1.5 Clase Lista Enlazada .....	16
2. Pruebas .....	19
2.1 Definición de JSON .....	19
2.2 Pruebas DFA .....	21
2.3 Pruebas de NFAe .....	23
2.4 Pruebas de ER .....	<b>¡Error! Marcador no definido.</b>

## Introducción

En el siguiente Api esta destinado para el desarrollo y la compresión de los autómatas finitos y determinísticos , junto a los no finitos y no determinísticos , y así también incluir las expresiones regulares. Comprender sus equivalencias, de uno a otro, saber y comprender las evaluaciones de expresiones en los autómatas y las conversiones de expresiones regulares a los autómatas no determinísticos y así como la equivalencia de los autómatas no determinísticos a los determinísticos para poder hacer sus respectivas evaluaciones.

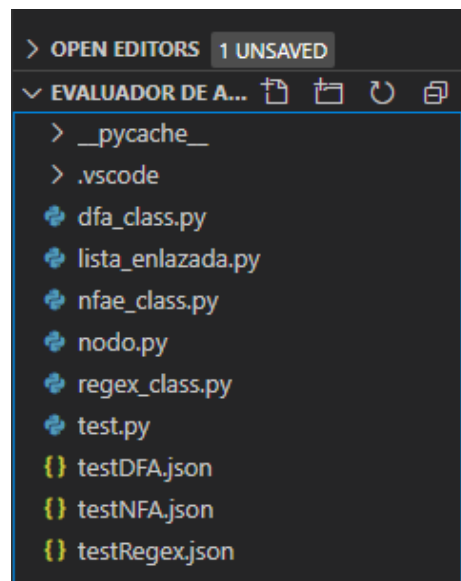
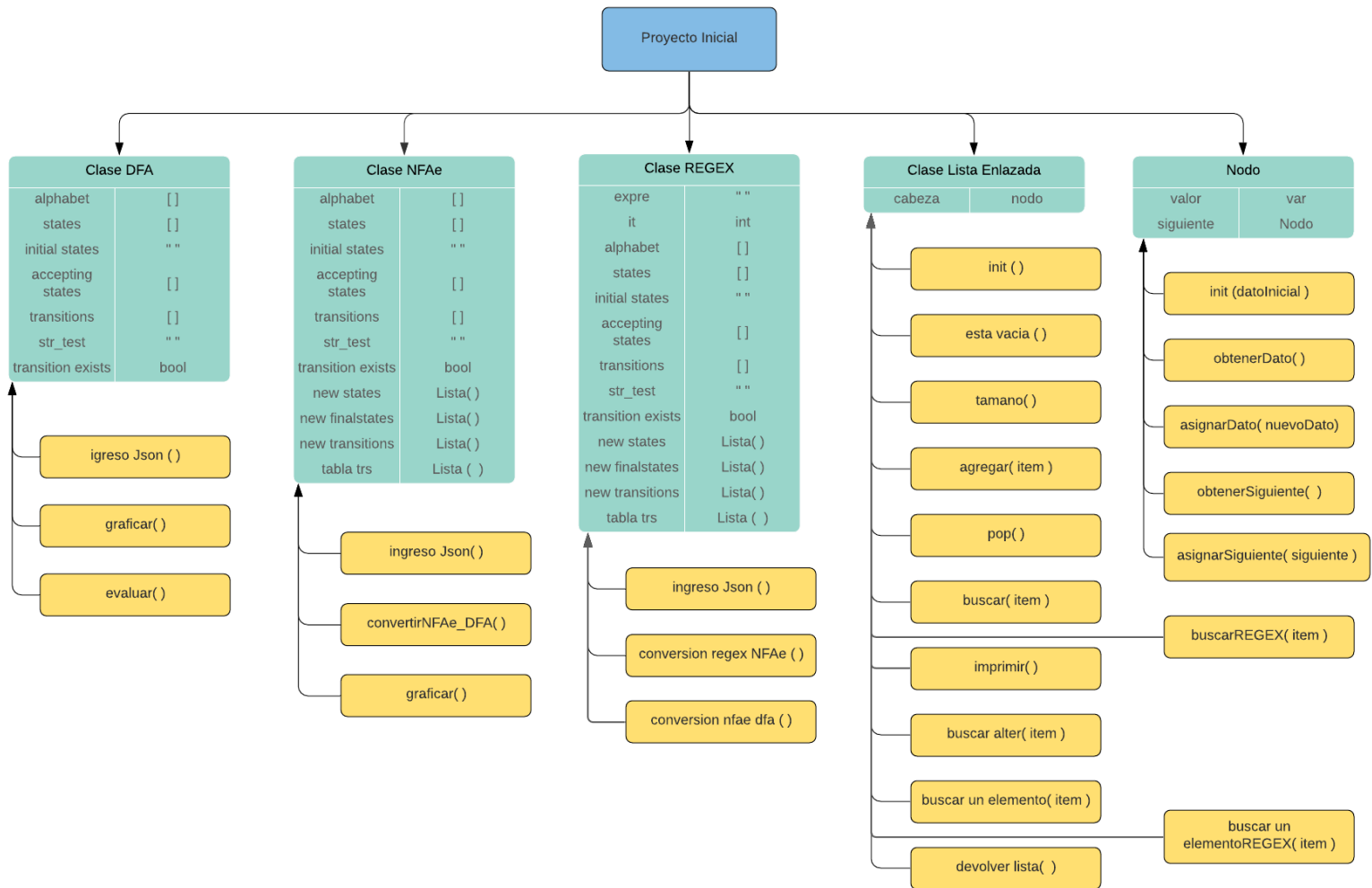
Poder mostrar el grafo resultante de cada uno de ellos con sus respectivos nodos los cuales serían los estados y poder representarles los mismos con sus transiciones. También poder explicar la finalidad de cada clase y dentro de la clase cada función utilizada.

# 1. Explicación de API y funciones Internas

## 1.1 Diagrama de Clases

### Automatas y Expresiones Regulares

Javier Nuñez | August 29, 2020



## 1.2 Clase DFA

Definición de clase DFA con sus atributos y funciones que cumplirán con los requerimientos definidos para el proyecto

```
dfa_class.py X
dfa_class.py > DFA
1 import json
2 import numpy as np
3 from os import system
4 import networkx as nx
5 import matplotlib.pyplot as plt
6
7 class DFA :
8     alphabet=[]
9     states =[]
10    initial_state=""
11    accepting_states =[]
12    transitions=[[0,0,0]]
13    str_test=""
14    transition_exists = True
15    cumple = True
16
17    #-----#
18    > def ingreso_json(self): ...
19
20    #-----#
21    > def evaluar(self): ...
22
23    #-----#
24    > def graficar(self): ...
25
26    #-----#
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
```

La cual quedaría así con sus atributos siendo los siguientes declarados, y las 3 funciones necesarias

Ingreso\_Json es la función que sirve para ingresar desde un archivo .json el cual debe estar en la carpeta del proyecto para sea posible hacerle la lectura, y siguiente a darle lectura ir igualando a sus atributos los atributos. Y en la misma función va evaluando si cumple con los requisitos para ser un autómata DFA

```
dfa_class.py X {} testDFA.json
dfa_class.py > DFA > ingreso_json
18 def ingreso_json(self):
19     if(self.cumple==True):
20         print("Archivo no compatible con DFA")
21         tmp = input("Nombre del archivo (.json): ")
22         try:
23             f= open(tmp,"r")
24             content = f.read()
25             jsondecoded = json.loads(content)
26             self.cumple = True
27             print("")
28             print("Mostrando Alfabeto")
29             for entity in jsondecoded["alphabet"]:
30                 if(entity=="E"):
31                     print("No se permite ingresar E en Alfabeto")
32                     break
33                 self.alphabet= np.append(self.alphabet, entity)
34             print(self.alphabet)
35             print("")
36             print("Mostrando Estados")
37             for entity in jsondecoded["states"]:
38                 self.states= np.append(self.states, entity)
39             print(self.states)
40             print("")
41             print("Mostrando Estado Inicial")
42             for entity in jsondecoded["initial_state"]:
43                 self.initial_state += entity
44             print(self.initial_state)
45             print("")
46             print("Mostrando Estados Finales")
47             for entity in jsondecoded["accepting_states"]:
48                 self.accepting_states= np.append(self.accepting_states, entity)
49             print(self.accepting_states)
50             print("")
51             print("Mostrando Transiciones")
52             for entity in jsondecoded["transitions"]:
53                 if(entity[1]=="E"):
54                     print("Transición con E no permitidas")
55                     self.cumple=False
56                     break
57                 self.transitions = np.append(self.transitions, [entity] , axis=0)
58             self.transitions = np.delete(self.transitions , 0 , axis=0)
59             for entity in jsondecoded["transitions"]:
60                 for x in self.transitions:
61                     if(entity[0]==x[0] and entity[1]==x[1] and entity[2]!= x[2]):
62                         self.cumple=False
63                         break
64             print(self.transitions)
65             if(self.cumple!=True):
```

```

65         if(self.cumple!=True):
66             system("cls")
67             self.alphabet=[]
68             self.states=[]
69             self.initial_state=""
70             self.accepting_states=[]
71             self.transitions=[[0,0,0]]
72             self.ingreso_json()
73         else:
74             print("Se Ingreso Correctamente")
75     except:
76         print("Archivo Invalido")
77         system("cls")
78         self.ingreso_json()

```

Evaluar esta función es en la que metemos la expresión desde la consola , al llamar la función desde el lugar que se llame , la función misma pide que el usuario ingrese la expresión que desea evaluar para saber si forma parte del lenguaje para el autómata que ingreso en la función anterior

Siguiente de pedir la expresión evalúa carácter por carácter la cadena de expresión ingresada y utilizando los atributos declarados para la clase, podemos recorrer nuestro autómata para saber si existe y le va mostrando en consola el recorrido de los caracteres y si es posible llegar a un punto de destino en cada uno, al ser posible retorna que Si pertenece de no ser posible No pertenece.

```

79 #-----#
80 def evaluar(self):
81     print("")
82     tmp = input("Valor a evaluar: ")
83     self.str_test=tmp
84     current_state = self.initial_state
85     transition_exists = True
86     for char_index in range(len(self.str_test)):
87         current_char = self.str_test[char_index]
88
89         for x in self.transitions:
90             if(x[0] == current_state and x[1] == current_char):
91                 transition_exists = True
92                 break
93             else:
94                 transition_exists = False
95         next_state=""
96         for x in self.transitions:
97             if(x[0]== current_state and x[1]==current_char):
98                 next_state=x[2]
99         print(current_state, current_char, next_state)
100         current_state = next_state
101     if transition_exists and current_state in self.accepting_states:
102         print("Pertenece a L(M)")
103     else:
104         print("No pertenece a L(M)")
105 #-----#
106 > def graficar(self): ...
128 #-----#

```

Graficar es la última función declarada en la clase de DFA la cual utiliza 2 diferentes librerías para lograr formar el grafo y mostrarlo en pantalla en un formato de imagen.png

```
Help
dfa_class.py - Evaluador de Automatas y convertido DFA NFA - Visual Studio Code

dfa_class.py X {} testDFA.json

dfa_class.py > DFA
1 import json
2 import numpy as np
3 from os import system
4 import networkx as nx
5 import matplotlib.pyplot as plt
6
7 class DFA:
8     alphabet=[]
9     states=[]
10    initial_state=""
11    accepting_states=[]
12    transitions=[[0,0,0]]
13    str_test=""
14    transition_exists = True
15    cumple = True
16
17    #-----#
18    > def ingreso_json(self):...
19    #-----#
20    > def evaluar(self):...
21    #-----#
22    def graficar(self):
23        print("")
24        print("Alphabet: ",self.alphabet)
25        print("States: ", self.states)
26        print("Initial State: ", self.initial_state)
27        print("Final States: ", self.accepting_states)
28        print("Transitions: \n", self.transitions)
29        grafo = nx.MultiDiGraph()
30
31        grafo.add_node(self.initial_state)
32
33        for x in self.states:
34            if(x != self.initial_state):
35                grafo.add_node(x)
36
37        for x in self.transitions:
38            grafo.add_edge(x[0], x[2], element=x[1])
39        print(grafo.edges())
40        nx.draw(grafo, with_labels=True)
41        plt.tight_layout()
42        plt.savefig("Grafo.png", format="PNG")
43        plt.show()
44    #-----#
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
```

### 1.3 Clase NFA-e

Para la clase NFA-e se declaro con los siguientes atributos, notamos que los atributos del autómata comparados con la clase anterior es una cantidad mayor y esto va por la función de conversión a NFA-e a DFA la cual es la más compleja de la clase compleja

```
nfae_class.py X
nfae_class.py > NFAe
1  import json
2  import numpy as np
3  import os
4  from os import system
5  import networkx as nx
6  import matplotlib.pyplot as plt
7  from lista_enlazada import Lista
8  from nodo import Nodo
9  class NFAe :
10     alphabet=[]
11     states=[]
12     initial_state=""
13     accepting_states=[]
14     transitions=[[0,0,0]]
15     str_test=""
16     transition_exists = True
17     new_states = Lista()
18     new_fstates = Lista()
19     new_transitions = Lista()
20     tabla_trs = Lista()
21
22     #-----#
23 > def ingreso_json(self): ...
24
25     #-----#
26 > def convertir_NFA_DFA(self): ...
27
28     #-----#
29 > def graficar(self): ...
30
31     #-----#
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
```

Primero declaramos la función de ingreso\_json la cual basa en la misma funcionalidad a la función ingreso\_json del autómata DFA, usando la misma lógica que el autómata DFA a excepción de las verificaciones con los requisitos de DFA

```
nfae_class.py X
nfae_class.py > NFAe
22 #-----#
23 def ingreso_json(self):
24     tmp = input("Valor a Test (.json): ")
25     try:
26         f= open(tmp , "r")
27         content = f.read()
28         jsondecoded = json.loads(content)
29
30         print("")
31         print("Mostrando Alfabeto")
32         for entity in jsondecoded["alphabet"]:
33             self.alphabet= np.append(self.alphabet, entity)
34         print(self.alphabet)
35
36         print("")
37         print("Mostrando Estados")
38         for entity in jsondecoded["states"]:
39             self.states= np.append(self.states, entity)
40         print(self.states)
41
42         print("")
43         print("Mostrando Estado Inicial")
44         for entity in jsondecoded["initial_state"]:
45             self.initial_state += entity
46         print(self.initial_state)
47
48         print("")
49         print("Mostrando Estados Finales")
50         for entity in jsondecoded["accepting_states"]:
51             self.accepting_states= np.append(self.accepting_states, entity)
52         print(self.accepting_states)
53
54         print("")
55         print("Mostrando Transiciones")
56         for entity in jsondecoded["transitions"]:
57             self.transitions = np.append(self.transitions, [entity] , axis=0)
58         self.transitions = np.delete(self.transitions , 0 , axis=0)
59         print(self.transitions)
60     except:
61         print("Archivo Invalido")
62         system("cls")
63         self.ingreso_json()
64 #-----#
```



Como siguiente función la de graficar NFA-e la cual expone la misma lógica que la función implementada en la clase DFA, quedando similar en código

```

nfae_class.py X
nfae_class.py > NFAe
22 #-----#
23 > def ingreso_json(self): ...
64 #-----#
65 > def convertir_NFA_DFA(self): ...
215 #-----#
216 def graficar(self):
217     print("")
218     print("Alphabet: ", self.alphabet)
219     print("States: ", self.states)
220     print("Initial State: ", self.initial_state)
221     print("Final States: ", self.accepting_states)
222     print("Transitions: \n", self.transitions)
223     grafo = nx.MultiDiGraph()
224
225     grafo.add_node(self.initial_state)
226
227     for x in self.states:
228         if x != self.initial_state:
229             grafo.add_node(x)
230
231     for x in self.transitions:
232         grafo.add_edge(x[0], x[2], element=x[1])
233
234     nx.draw(grafo, with_labels=True)
235     plt.tight_layout()
236     plt.savefig("Grafo.png", format="PNG")
237     plt.show()
238 #-----#
239

```

Y como última función implemente la función convertir\_NFAe\_DFA la cual como dije declare anteriormente es la mas compleja de la clase NFAe siendo el código el siguiente:

```

nfae_class.py X
nfae_class.py > NFAe > convertir_NFA_DFA
65 def convertir_NFA_DFA(self):
66     #dfa = DFA()
67     print("#----- Convirtiendo -----#")
68     cerraduraE = []
69     cerraduraD = []
70     cerraduraEF = []
71     for al in self.alphabet:
72         if al != "E":
73             for es in self.states:
74                 cerraduraE = np.append(cerraduraE, [es])
75             for tra in self.transitions:
76                 if (es == tra[0] and tra[1] == "E"):
77                     cerraduraE = np.append(cerraduraE, [tra[2]])
78             for est in cerraduraE:
79                 for trs in self.transitions:
80                     if (est == trs[0] and trs[1] == al):
81                         cerraduraD = np.append(cerraduraD, [trs[2]], axis=0)
82             for estad in cerraduraD:
83                 cerraduraEF = np.append(cerraduraEF, [estad])
84             for tran in self.transitions:
85                 if (estad == tran[0] and tran[1] == "E"):
86                     cerraduraEF = np.append(cerraduraEF, [tran[2]])
87             if (cerraduraEF != []):
88                 if (self.new_states.buscar(cerraduraEF) == False):
89                     self.new_states.agregar(cerraduraEF)
90                 aux = list(cerraduraEF)
91                 self.tabla_trs.agregar([es], [al], aux)
92                 cerraduraEF = []
93
94             cerraduraD = []
95             cerraduraE = []
96         else:
97             print("Epsilon")
98     self.new_states.agregar([self.initial_state])
99     for x in self.accepting_states:
100         self.new_fstates = self.new_states.buscar_alter(x)
101     iterador=0
102     tamaño=self.tabla_trs.tamaño()
103     while(iterador< tamaño):
104         tmp = self.tabla_trs.pop()
105         aux = np.array(tmp)
106         if(tmp[2] != [] and not(self.new_transitions.buscar(aux))):
107             aux = np.array(tmp[0])
108             if(self.new_states.buscar(aux)):
109                 self.new_transitions.agregar(tmp)
110             iterador= iterador+1
111     lista_aux = self.new_transitions.devolverLista()
112     iterador=0
113     tamaño=self.new_transitions.tamaño()

```

```

nfae_class.py X
nfae_class.py > NFAe > convertir_NFA_DFA

114 while(iterator< tamano):
115     tmp = lista_aux.pop()
116     aux = np.array(tmp)
117     aux = np.array(aux[2])
118     if not(self.new_transitions.buscar_uno_en_elemento(aux ,0)):
119         if(len(aux)>1):
120             arr = []
121             for al in self.alphabet:
122                 if not(al == "E"):
123                     for y in aux:
124                         for x in self.transitions:
125                             if(x[0]== y) and (x[1] == al):
126                                 arr = np.append(arr , x[2])
127                                 aux_tmp = [ list(aux) , al , list(arr)]
128                                 self.new_transitions.agregar(aux_tmp)
129                                 arr=[]
130             else:
131                 for al in self.alphabet:
132                     for x in self.transitions:
133                         if(x[0]== aux) and (x[1] == al):
134                             aux_tmp = [ aux , al , x[2]]
135                             self.new_transitions.agregar(aux_tmp)
136         iterator= iterator+1
137     print("#----- Graficando -----#")
138     print("")
139     self.alphabet = np.delete(self.alphabet , 0)
140     print("Alphabet: ",self.alphabet)
141     print("States: ")
142     self.new_states.imprimir()
143     print("Initial State: ", self.initial_state)
144     print("Final States: ")
145     self.new_fstates.imprimir()
146     print("\nTransitions:")
147     self.new_transitions.imprimir()
148     grafo = nx.MultiDiGraph()
149     grafo.add_node(self.initial_state)
150     tam = self.new_states.tamano()
151     while(tam>0):
152         x= self.new_states.pop()
153         aux = np.append(aux , x)
154         if not((aux == [self.initial_state]).all()):
155             listToStr = ' '.join(x)
156             grafo.add_node(listToStr)
157             tam = tam - 1
158     tam = self.new_transitions.tamano()
159     lista_aux = self.new_transitions.devolverLista()
160     while(tam>0):
161         x= lista_aux.pop()
162         s = ' '.join(x[0])

```

```

nfae_class.py X
nfae_class.py > NFAe

162     s = ' '.join(x[0])
163     d = ' '.join(x[2])
164     a = ' '.join(x[1])
165     grafo.add_edge(s, d, element=a)
166     tam = tam - 1
167     nx.draw(grafo , with_labels=True)
168     plt.tight_layout()
169     plt.savefig("Grafo.png", format="PNG")
170     plt.show()
171     print("#----- Evaluando -----#")
172     print("")
173     tmp = input("Valor a evaluar: ")
174     self.str_test = tmp
175     current_state= np.array(self.initial_state)
176     for char_index in range(len(self.str_test)):
177         lista_aux = self.new_transitions.devolverLista()
178         current_char = self.str_test[char_index]
179         tam = self.new_transitions.tamano()
180         while (tam>0):
181             x = list(lista_aux.pop())
182             aux= np.array(x[0])
183             current_state=np.array(current_state)
184             if ((aux==current_state).all()) and (x[1] == current_char):
185                 self.transition_exists = True
186                 break
187             else:
188                 self.transition_exists = False
189             tam = tam -1
190         lista_aux = self.new_transitions.devolverLista()
191         tam = self.new_transitions.tamano()
192         while (tam>0):
193             x = list(lista_aux.pop())
194             aux= np.array(x[0])
195             aux2 = np.array(x[2])
196             aux3 = np.array(current_char)
197             if((aux == current_state ).all() and x[1] == aux3):
198                 next_state = np.array(aux2)
199                 break
200             tam = tam-1
201         print(current_state, current_char, next_state)
202         current_state = next_state
203     current_state= np.array(current_state)
204     if (self.new_fstates.buscar(current_state)):
205         print("Pertenece a L(M)")
206     else:
207         print("No pertenece a L(M)")

```

Para la explicación de esta función , sería en dos partes , la explicación de esta función junto con la explicación de la clase de Listas Enlazadas

La función se despliega en 3 partes:

- La primera parte es en la que se recorren las transiciones actuales para poder sacar las C\_E (cerraduras épsilon), seguido de las D(C\_E) (Transiciones de cerradura épsilon) y por último la C\_EF que sería la cerradura épsilon final la cual nos servirá para sacar la tabla de transiciones de nuestro autómata transformado a DFA, junto con los nuevos estados que estarían ligados a nuestro nuevo autómata DFA

Ahí vienen las Listas enlazadas utilizadas para guardar todas las transiciones en forma de arreglos y aparte de las transiciones los estados y los estados de aceptación.

Todo esto se va haciendo en un for en cual saca solo los valores del alfabeto del autómata que no sea E(épsilon). Ahí aclaro se va haciendo primero para ejemplo a luego hace todas las cerraduras para b y así sucesivamente hasta que ya no encuentre más valores del alfabeto.

- Como segunda parte utilizando las listas enlazadas y sus propiedades , teniendo ya el autómata convertido a DFA , procedo a graficarlo en la misma función utilizando siempre las mismas librerías y la misma lógica para graficar que utilice en las funciones anteriores, pero utilizando las listas.
- Seguido en la misma función procedo a pregunta una expresión para evaluar en el autómata convertido , ya que en el NFAe no se puede evaluar entonces en la misma función pido una expresión para evaluación. Utilizando igual la misma lógica, pero con el código diferente que utiliza listas en vez de arreglos numpy.

## 1.4 Clase REGEX (Expresiones Regulares)

La clase de expresiones Regulares esta definida de la siguiente manera tal que pueda reutilizar de cierta forma el código hecho en la clase NFAe para hacer la equivalencia a DFA la cual llevaría incluida la manera de evaluación en la misma. Definida de la siguiente forma:

```
test.py  regex_class.py  lista_enlazada.py
regex_class.py > Regex
1  import os
2  import json
3  import numpy as np
4  import networkx as nx
5  import matplotlib.pyplot as plt
6
7  from os import system
8  from nfae_class import NFAe
9  from lista_enlazada import Lista
10 from nodo import Nodo
11
12 class Regex :
13     expre= ""
14     it=1
15     alphabet=[]
16     states=[]
17     initial_state="q0"
18     accepting_states=[]
19     operations=[]
20     transitions=[[0,0,0,0]]#[estadosaliente , elemento alfabeto , estado destino , flag inicial]
21     new_states = Lista()
22     new_fstates = Lista()
23     new_transitions = Lista()
24     tabla_trs = Lista()
25
26 > def ingreso_json(self): ...
41
42 > def conversion_regex_NFAe(self): ...
123
124 > def conversion_nfae_dfa(self): ...
```

Definiría una función ingreso de json el cual estaría con la misma funcionalidad de las demás clases, el cual constaría simplemente de leer un json el cual contenga la expresión regular a utilizar y transformar y en la cual se evalúa si es una expresión valida.

```
25
26 def ingreso_json(self):
27     tmp = input("Valor a Test (.json): ")
28     try:
29         f= open(tmp , "r")
30         content = f.read()
31         jsondecoded = json.loads(content)
32         print("")
33         print("Mostrando Expresion")
34         for entity in jsondecoded["expresion"]:
35             self.expre += entity
36             print(self.expre)
37
38     except:
39         system("cls")
40         self.ingreso_json()
41
```

Como siguiente función definida tengo `convertir_regex_NFAe()` la cual se encarga de hacer la respectiva transformación o equivalencia de la expresión regular hacia el NFAe correspondiente para la misma. Basándome en código, utiliza una bandera para saber cuándo una expresión lleva `|` `*` `+` utilizada de forma que con la misma puedo saber que transición va antes y después de cada operación y con ello pude sacar cada transición que se debería hacer suponiendo que `q0` siempre sería una transición con E al primer estado o primeros estados dependiendo de las operaciones y un arreglo temporal utilizado con el mismo propósito pero al inversa saber cual es el ultimo estado de las transiciones que son ultimas , para saber qué estado tendría la transición con E a un `qFinal` , explicado ello , el código de la función sería el siguiente:

```

42 def conversion_regex_NFAe(self):
43     ini= 0
44     tmpFinales=[]
45     est_i=""
46     est_f=""
47     lastC=""
48     for char in self.expres:
49         if char!=' ':
50             if char == '|':
51                 self.operations = np.append(self.operations , [char])
52                 est_f= "q"+ str(self.it)
53                 tmpFinales = np.append(tmpFinales , est_f)
54                 self.it+=1
55                 ini=0
56             elif char=='*':
57                 self.operations = np.append(self.operations , [char])
58                 trns = [est_f , lastC , est_i,ini]
59                 self.transitions = np.append(self.transitions , [trns] , axis=0)
60             elif char=='+' :
61                 est_f= "q"+ str(self.it)
62                 self.it+=1
63                 self.operations = np.append(self.operations , [char])
64                 auxInt= len(self.transitions)
65                 if(self.transitions[auxInt-1][3]!='0'):
66                     ini=0
67                 self.transitions = np.delete(self.transitions, (auxInt-1), axis=0)
68                 trns = [est_i , lastC , est_i,ini]
69                 self.transitions = np.append(self.transitions , [trns] , axis=0)
70                 self.states = np.delete(self.states , (len(self.states)-1), axis=0)
71                 ini=1
72             else:
73                 if char not in self.alphabet:
74                     self.alphabet = np.append(self.alphabet , [char])
75                     est_i= "q"+ str(self.it)
76                     self.it+=1
77                     est_f= "q"+ str(self.it)
78                     trns = [est_i , char , est_f,ini]
79                     lastC=char
80                     self.transitions = np.append(self.transitions , [trns] , axis=0)
81                     ini=1
82                 if est_i not in self.states:
83                     self.states = np.append(self.states , [est_i] , axis=0)
84                 if est_f not in self.states:
85                     self.states = np.append(self.states , [est_f] , axis=0)
86             est_i= "q0"
87             est_f= "q"+ str(self.it)
88             tmpFinales = np.append(tmpFinales , est_f)

```

```

91 self.transitions = np.delete(self.transitions , 0 , axis=0)
92 for trs in self.transitions:
93     aux = list(trs)
94     if aux[3]=='0':
95         trns = [est_i , 'E' , aux[0] , 1 ]
96         self.transitions = np.append(self.transitions , [trns] , axis=0)
97
98     if '|' in self.operations:
99         self.it+=1
100         est_f= "q"+ str(self.it)
101         for st in tmpFinales:
102             trns = [st , 'E' , est_f, 1]
103             self.transitions = np.append(self.transitions , [trns] , axis=0)
104     else:
105         est_i= "q"+ str(self.it)
106         self.it+=1
107         est_f= "q"+ str(self.it)
108         trns = [tmpFinales[0] , 'E' , est_f, 1]
109         self.transitions = np.append(self.transitions , [trns] , axis=0)
110
111 self.states = np.append(self.states , [est_f] , axis=0)
112 self.states = np.insert(self.states , 0, ["q0"])
113 self.alphabet = np.insert(self.alphabet, 0 , ['E'])
114 self.transitions = np.delete(self.transitions , 3 , axis=1)
115 self.accepting_states = np.array([est_f])
116
117 print("")
118 print("Mostrando Alfabeto")
119 print(self.alphabet)
120 print("Mostrando Estados")
121 print(self.states)
122 print("Mostrando Estado Inicial")
123 print(self.initial_state)
124 print("Mostrando Estados Finales")
125 print(self.accepting_states)
126 print("Mostrando Operaciones")
127 print(self.operations)
128 print("Transiciones :")
129 print(self.transitions)
130
131 grafo = nx.MultiDiGraph()
132 grafo.add_node(self.initial_state)
133 for x in self.states:
134     if(x != self.initial_state):
135         grafo.add_node(x)
136
137 for x in self.transitions:
138     grafo.add_edge(x[0], x[2], element=x[1])
139

```

En la parte final de la función no termino de poner las líneas de código faltante que es donde se manda a llamar la librería que esta encargada de hacer la imagen png y almacenarla en la carpeta del proyecto, líneas las cuales se pueden apreciar mejor en las funciones graficar de la clase DFA.

Y por ultimo ya que no pude utilizar la función de NFae, reescribi la función `convertir_nfae_dfa()` para que quedara con las necesidades que me pedia el NFae resultante de la función `convertir_regex_NFAe()` pequeños cambios hecho , ciertas validaciones de mas y junto a ello implemente cambios en algunas funciones de mi clase `lista_enlazada` para poder cumplir los requisitos, al finalizar la función me queda el código resultante siguiente:

```

145 def conversion_nfae_dfa(self):
146     print("#----- Convirtiendo -----#")
147     cerraduraE = []
148     cerraduraD = []
149     cerraduraEF=[]
150     for al in self.alphabet:
151         if(al!="E"):
152             for es in self.states:
153                 cerraduraE = np.append(cerraduraE , [es])
154                 for tra in self.transitions:
155                     if(es == tra[0] and tra[1] == "E"):
156                         cerraduraE = np.append(cerraduraE , [tra[2]])
157                 for est in cerraduraE:
158                     for trs in self.transitions:
159                         if(est == trs[0] and trs[1]==al):
160                             cerraduraD = np.append(cerraduraD , [trs[2]] , axis=0)
161                 for estad in cerraduraD:
162                     cerraduraEF = np.append(cerraduraEF , [estad])
163                     for tran in self.transitions:
164                         if(estad == tran[0] and tran[1] == "E"):
165                             cerraduraEF = np.append(cerraduraEF , [tran[2]])
166             if not(len(cerraduraEF) == 0):
167                 if(self.new_states.buscarREGEX(cerraduraEF)==False):
168                     self.new_states.agregar(cerraduraEF)
169                     aux = list(cerraduraEF)
170                     self.tabla_trs.agregar([[es] , [al] , aux])
171             cerraduraEF=[]
172         else:
173             cerraduraD=[]
174             cerraduraE=[]
175     else:
176         print("")
177     self.new_states.agregar([self.initial_state])
178     listatmp=Lista()
179     # ESTADOS FINALES NUEVOS
180     for x in self.accepting_states:
181         listatmp = self.new_states.buscar_alter(x)
182         i = listatmp.tamano()
183         while (i >0):
184             tmp = listatmp.pop()
185             self.new_fstates.agregar(tmp)
186             i=i-1
187     #TRANSICIONES NUEVAS
188     iterador=0
189     tamaño=self.tabla_trs.tamano()
190     lista_aux=self.tabla_trs.devolverLista()
191     while(iterador< tamaño):
192         tmp = lista_aux.pop()
193         aux = np.array(tmp,dtype=object)

```

```
regex_class.py X
regex_class.py
194         if not(len(tmp[2])==0) and not(self.new_transitions.buscar(aux)):
195             aux = np.array(tmp[0])
196             if(self.new_states.buscar(aux)):
197                 self.new_transitions.agregar(tmp)
198             iterador= iterador+1
199 lista_aux = self.new_transitions.devolverLista()
200 iterador=0
201 tamaño=self.new_transitions.tamaño()
202 while(iterador< tamaño):
203     tmp = lista_aux.pop()
204     aux = np.array(tmp , dtype=object)
205     aux = np.array(aux[2])
206     if not(self.new_transitions.buscar_uno_en_elementoREGEX(aux ,0)):
207         if(len(aux)>1):
208             arr = []
209             for al in self.alphabet:
210                 if not(al == "E"):
211                     for y in aux:
212                         for x in self.transitions:
213                             if(x[0]== y) and (x[1] == al):
214                                 arr = np.append(arr , x[2])
215             aux_tmp = [ list(aux) , al , list(arr)]
216             if not(len(aux_tmp[2])==0) and self.new_states.buscarREGEX(aux_tmp[2]):
217                 self.new_transitions.agregar(aux_tmp)
218             arr=[]
219         else:
220             for al in self.alphabet:
221                 if not(al=="E"):
222                     for x in self.transitions:
223                         if(x[0]== aux) and (x[1] == al):
224                             aux_tmp = [ aux , al , x[2]]
225                             if not(len(aux_tmp[2])==0) and self.new_states.buscarREGEX(x[2]):
226                                 self.new_transitions.agregar(aux_tmp)
227 iterador= iterador+1
228
```

Cabe destacar que en las funciones anteriores de conversiones ya va incluido lo que es la manera de graficar y mostrar el grafo en las funciones , y mostraría un grafo para cada función.

**Nota:** cada grafo tiene una imagen almacenada en la carpeta del proyecto con su respectivo nombre que define que se hizo en el grafo.

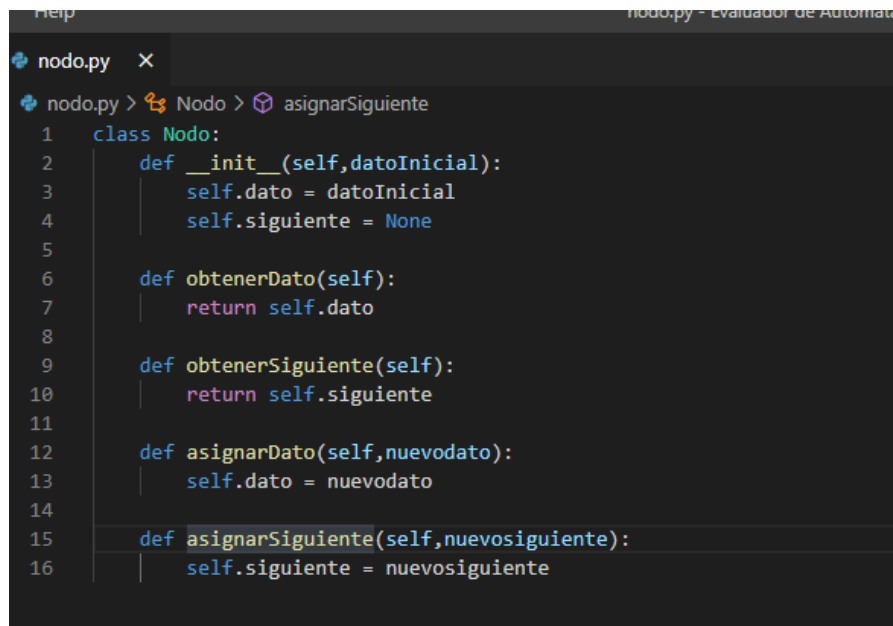


## 1.5 Clase Lista Enlazada

En mi clase de lista enlazada utilizo una lista enlazada normal con su clase de nodo el cual se utiliza en mi caso para guardar el valor necesario para la lista.

En ella declaro ciertas funciones aparte de las que están por defecto en la lista enlazada que todos conocemos, las cuales explicare a continuación:

Clase Nodo:



```
Help
nodo.py - Evaluador de Automata

nodo.py > Nodo > asignarSiguiente

1 class Nodo:
2     def __init__(self,datoInicial):
3         self.dato = datoInicial
4         self.siguiente = None
5
6     def obtenerDato(self):
7         return self.dato
8
9     def obtenerSiguiente(self):
10        return self.siguiente
11
12    def asignarDato(self,nuevodato):
13        self.dato = nuevodato
14
15    def asignarSiguiente(self,nuevosiguiente):
16        self.siguiente = nuevosiguiente
```

Clase Lista Enlazada:



```
lista_enlazada.py X
lista_enlazada.py
1 from nodo import Nodo
2 import numpy as np
3
4 class Lista:
5
6 > def __init__(self): ...
7
8
9 > def estaVacia(self): ...
10
11
12 > def agregar(self,item): ...
13
14
15
16
17 > def tamaño(self): ...
18
19
20
21
22
23
24
25 > def pop(self): ...
26
27
28
29
30
31
32 > def imprimir(self): ...
33
34
35
36
37
38 > def buscar(self,item): ...
39
40
41
42
43
44
45
46
47
48 > def buscarREGEX(self,item): ...
49
50
51
52
53
54
55
56
57
58 > def buscar_alter(self,item): ...
59
60
61
62
63
64
65
66
67
68
69
70 > def buscar_uno_en_elemento(self,item ,i): ...
71
72
73
74
75
76
77
78
79
80
81 > def buscar_uno_en_elementoREGEX(self,item ,i): ...
82
83
84
85
86
87
88
89
90
91
92 > def devolverLista(self): ...
```

Las funciones por defecto son las init, está Vacía , agregar , tamaño, buscar e imprimir las cuales son las que una lista enlazada normalmente contiene.

La función pop es una función en la cual implemento la lógica en el código que me retorne la cabeza de la lista enlazada y quita la cabeza de la lista enlazada poniendo su valor siguiente como cabeza.

```
24
25     def pop(self):
26         if (self.cabeza == None):
27             return None
28         tmp = self.cabeza
29         self.cabeza = tmp.obtenerSiguiente()
30         return tmp.obtenerDato()
31
```

La función buscar\_alter es una función que declaro para hacer una búsqueda alternativa lo que hace básicamente es buscar un valor en la lista dentro de los nodos en los arreglos y retorna una lista con los valores que le contengan el ítem enviado

```
48     def buscar_alter(self,item):
49         actual = self.cabeza
50         lista = Lista()
51         tmp = []
52         while actual != None:
53             tmp = actual.obtenerDato()
54             for x in tmp:
55                 if(x == item):
56                     lista.agregar(actual.obtenerDato())
57             actual= actual.obtenerSiguiente()
58         return lista
59
60
```

La función buscar\_uno\_en\_elemento es una función parecida a buscar, pero lo que hace es devolver Verdadero si el ítem existe dentro del arreglo en el elemento nodo

Y la función devolver lista es una función auxiliar que utilizo para devolver toda la lista completa para hacer asignaciones a otras listas, por ejemplo.

```
60
61     def buscar_uno_en_elemento(self,item ,i):
62         actual = self.cabeza
63         encontrado = False
64         while actual != None and not encontrado:
65             aux= actual.obtenerDato()
66             if ((aux[i] == item).all()):
67                 encontrado = True
68             else:
69                 actual = actual.obtenerSiguiente()
70         return encontrado
71
72     def devolverLista(self):
73         actual = self.cabeza
74         lista = Lista()
75         while( actual != None):
76             lista.agregar(a= actual.obtenerDato() , obtenerSiguiente: obtenerSiguiente)
77             actual= actual.obtenerSiguiente()
78         return lista
```

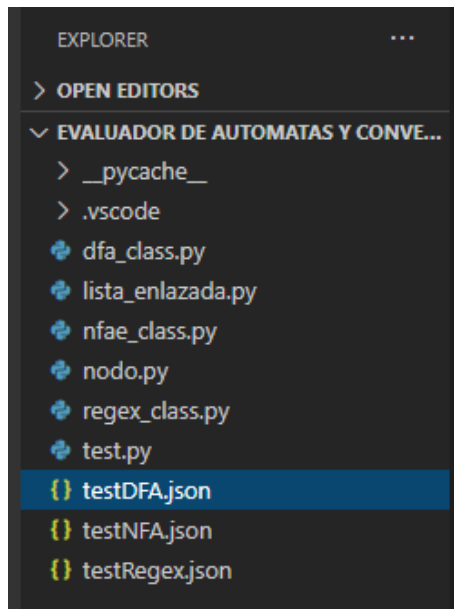
Declare al final 2 funciones mas especialmente para ser utilizadas por la clase REGEX la cual por cuestiones de los arreglos que estaba utilizando para almacenar los datos me tiraba errores de inconsistencia de formas, ósea que no eran de forma regular como de costumbre usaba para comparar , lo cual me llevo a utilizar el recurso de crear nuevas funciones las cuales cumplieran con esos requerimientos , hacen lo mismo que sus funciones originales con el cambio de que estas son orientada a elemento list con el cual si podía hacer las comparaciones necesarias.

```
47
48 def buscarREGEX(self,item):
49     actual = self.cabeza
50     encontrado = False
51     while actual != None and not encontrado:
52         if (list(actual.obtenerDato()) == list(item)):
53             encontrado = True
54         else:
55             actual = actual.obtenerSiguiente()
56     return encontrado
57
58 > def buscar_alter(self,item): ...
59
70 > def buscar_uno_en_elemento(self,item ,i): ...
80
81 def buscar_uno_en_elementoREGEX(self,item ,i):
82     actual = self.cabeza
83     encontrado = False
84     while (actual != None and not encontrado):
85         aux = actual.obtenerDato()
86         if (list(aux[i]) == list(item)):
87             encontrado = True
88         else:
89             actual = actual.obtenerSiguiente()
90     return encontrado
```

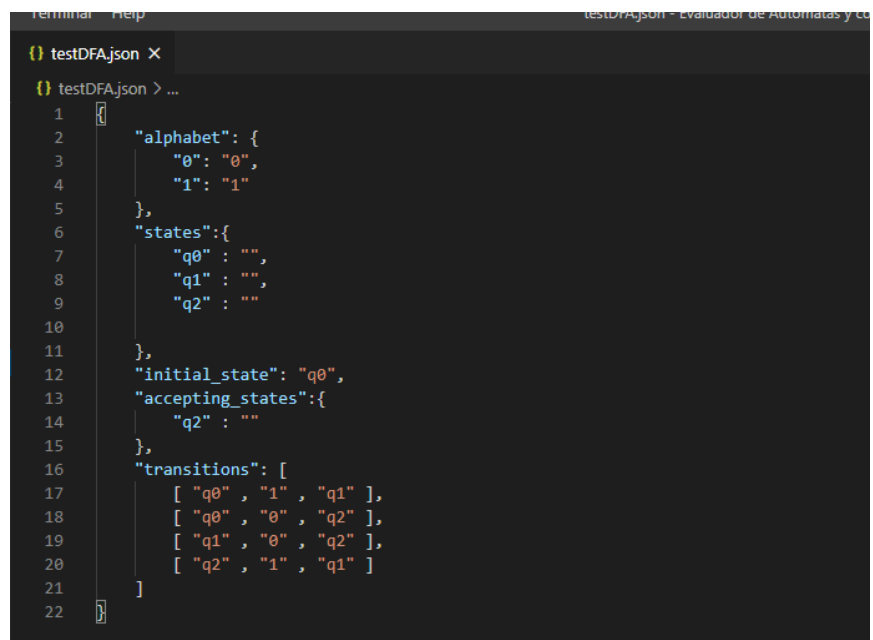
## 2. Pruebas

### 2.1 Definición de JSON

Para mis pruebas utilice los siguientes archivos con extensión json uno para cada autómatata y uno para las expresiones regulares. Estos se definirían de la siguiente forma:



Para un DFA:



Para un NFAe:

```
testNFA.json X
testNFA.json > ...
1 {
2   "alphabet": {
3     "E": "E",
4     "a": "a",
5     "b": "b"
6   },
7   "states": {
8     "q0": "",
9     "q1": "",
10    "q2": "",
11    "q3": "",
12    "q4": "",
13    "q5": ""
14  },
15  "initial_state": "q0",
16  "accepting_states": {
17    "q5": ""
18  },
19  "transitions": [
20    [ "q0", "E", "q1"],
21    [ "q0", "E", "q2"],
22    [ "q1", "b", "q3"],
23    [ "q2", "a", "q4"],
24    [ "q3", "E", "q5"],
25    [ "q4", "E", "q5"],
26    [ "q5", "b", "q5"],
27    [ "q5", "a", "q5"]
28  ]
29 }
30 }
```

Y finalmente el mas sencillo que es para una Expresión Regular:

```
testRegex.json X
testRegex.json > expression
1 {
2   "expression": "ab+ | b*"
3 }
```

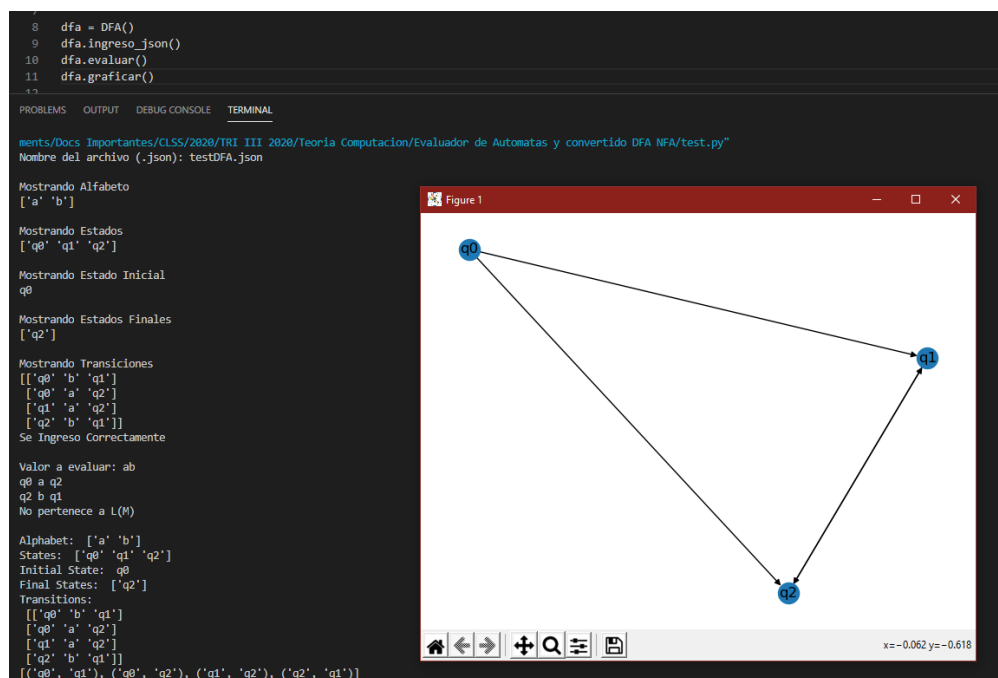
## 2.2 Pruebas DFA

Para las pruebas de DFA hice 3 diferentes pruebas:

Prueba 1:

```
testDFA.json X test.py
testDFA.json > [ ] transitions
1 {
2   "alphabet": {
3     "a": "a",
4     "b": "b"
5   },
6   "states":{
7     "q0" : "",
8     "q1" : "",
9     "q2" : ""
10  },
11  "initial_state": "q0",
12  "accepting_states":{
13    "q2" : ""
14  },
15  "transitions": [
16    [ "q0", "b", "q1" ],
17    [ "q0", "a", "q2" ],
18    [ "q1", "a", "q2" ],
19    [ "q2", "b", "q1" ]
20  ]
21 }
22 }
```

Resultados:



## Prueba 2:

```

testDFA.json x test.py
testDFA.json > [ ] transitions > [ ] 4 > 1
1 {
2   "alphabet": {
3     "a": "a",
4     "b": "b",
5     "c": "c"
6   },
7   "states": {
8     "q0": "",
9     "q1": "",
10    "q2": "",
11    "q3": ""
12  },
13  },
14  "initial_state": "q0",
15  "accepting_states": {
16    "q3": ""
17  },
18  "transitions": [
19    [ "q0", "b", "q1" ],
20    [ "q0", "a", "q2" ],
21    [ "q1", "a", "q2" ],
22    [ "q2", "b", "q1" ],
23    [ "q2", "c", "q3" ]
24  ]
25 }

```

## Resultados:

```

8 dfa = DFA()
9 dfa.ingreso_json()
10 dfa.evaluar()
11 dfa.graficar()
12

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL**

Nombre del archivo (.json): testDFA.json

Mostrando Alfabeto  
['a' 'b' 'c']

Mostrando Estados  
['q0' 'q1' 'q2' 'q3']

Mostrando Estado Inicial  
q0

Mostrando Estados Finales  
['q3']

Mostrando Transiciones  
[['q0' 'b' 'q1']  
['q0' 'a' 'q2']  
['q1' 'a' 'q2']  
['q2' 'b' 'q1']  
['q2' 'c' 'q3']]

Se Ingreso Correctamente

Valor a evaluar: babac  
q0 b q1  
q1 a q2  
q2 b q1  
q1 a q2  
q2 c q3  
Pertenece a L(M)

Alphabet: ['a' 'b' 'c']  
States: ['q0' 'q1' 'q2' 'q3']  
Initial State: q0  
Final States: ['q3']  
Transitions:  
[['q0' 'b' 'q1']  
['q0' 'a' 'q2']  
['q1' 'a' 'q2']  
['q2' 'b' 'q1']  
['q2' 'c' 'q3']]

[('q0', 'q1'), ('q0', 'q2'), ('q1', 'q2'), ('q2', 'q1'), ('q2', 'q3')]

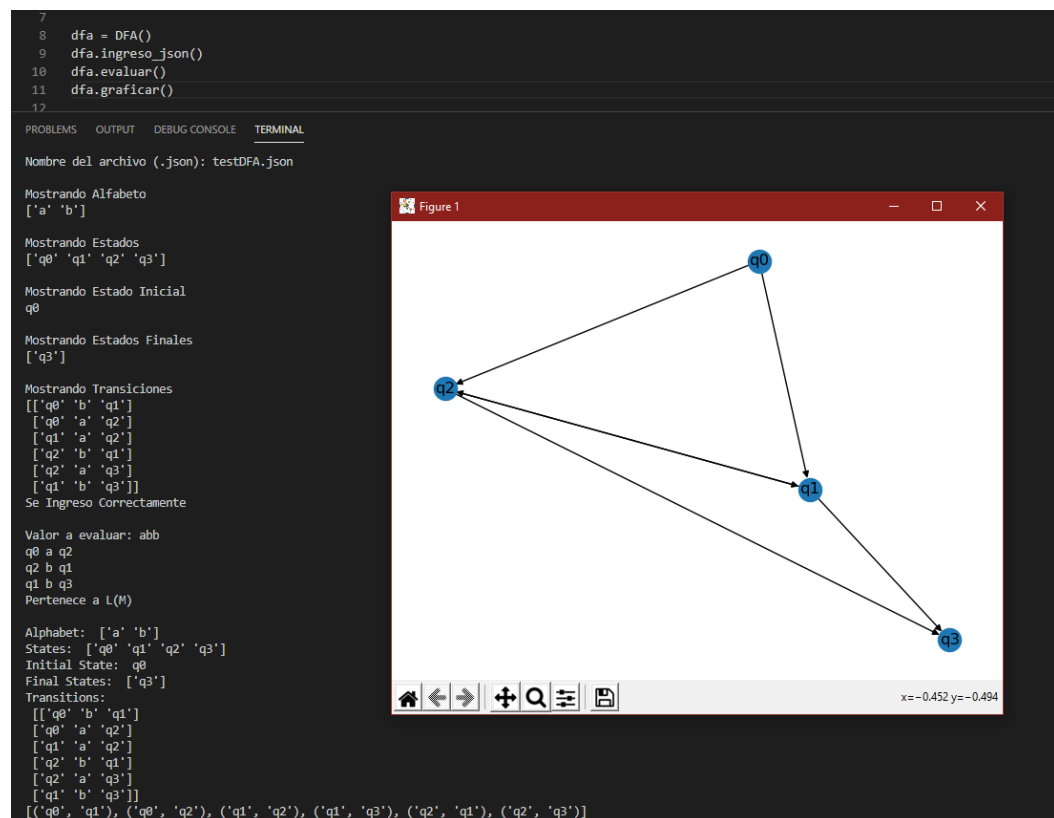
Figure 1

x=-0.091 y=0.043

### Prueba 3:

```
testDFAjson X test.py
testDFAjson > [ ] transitions > [ ] 5
1 {
2   "alphabet": {
3     "a": "a",
4     "b": "b"
5   },
6   "states":{
7     "q0" : "",
8     "q1" : "",
9     "q2" : "",
10    "q3" : ""
11  },
12  },
13  "initial_state": "q0",
14  "accepting_states":{
15    "q3" : ""
16  },
17  "transitions": [
18    [ "q0", "b", "q1" ],
19    [ "q0", "a", "q2" ],
20    [ "q1", "a", "q2" ],
21    [ "q2", "b", "q1" ],
22    [ "q2", "a", "q3" ],
23    [ "q1", "b", "q3" ]
24  ]
25 }
```

### Resultados:





## 2.3 Prueba NFA

Prueba 1:

```
test.py  testNFA.json x
testNFA.json > ...
1  [
2    "alphabet": {
3      "E": "E",
4      "a": "a",
5      "b": "b"
6    },
7    "states": {
8      "q0": "",
9      "q1": "",
10     "q2": "",
11     "q3": "",
12     "q4": "",
13     "q5": ""
14   },
15   },
16   "initial_state": "q0",
17   "accepting_states": {
18     "q5": ""
19   },
20   "transitions": [
21     [ "q0", "E", "q1"],
22     [ "q0", "E", "q2"],
23     [ "q1", "b", "q3"],
24     [ "q2", "a", "q4"],
25     [ "q3", "E", "q5"],
26     [ "q4", "E", "q5"],
27     [ "q5", "b", "q5"],
28     [ "q5", "a", "q5"]
29   ]
30 ]
```

Resultado:

```
12
13 nfae = NFAe()
14 nfae.ingreso_json()
15 nfae.graficar()
16 nfae.convertir_NFAe_DFA()
17
18 #regex = Regex()
19 #regex.ingreso_json()
20 #nfae.convertir_nfae_dfa()

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
nfae/Docs Importantes/CLSS/2020/TRI III 2020/Teoria Computacion/Evaluador de Automatas y convertido DFA NFA/test.py
Valor a Test (.json): testNFA.json

Mostrando Alfabeto
['E' 'a' 'b']

Mostrando Estados
['q0' 'q1' 'q2' 'q3' 'q4' 'q5']

Mostrando Estado Inicial
q0

Mostrando Estados Finales
['q5']

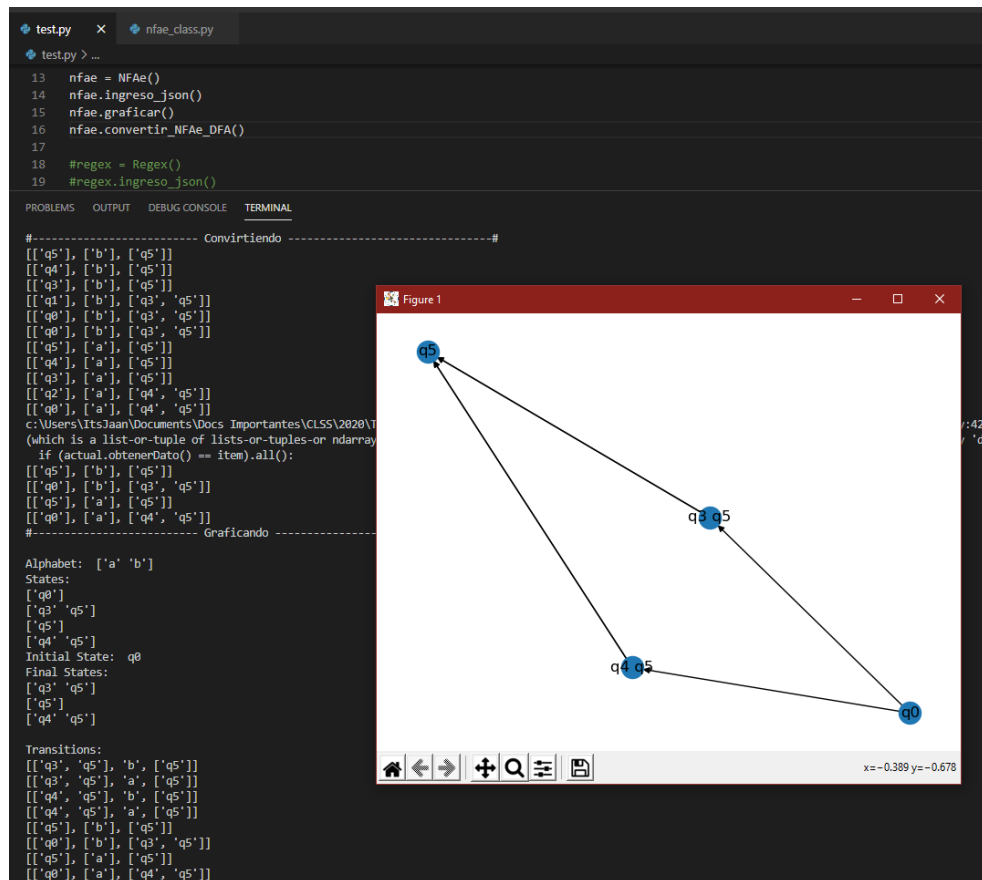
Mostrando Transiciones
[['q0' 'E' 'q1']
 ['q0' 'E' 'q2']
 ['q1' 'b' 'q3']
 ['q2' 'a' 'q4']
 ['q3' 'E' 'q5']
 ['q4' 'E' 'q5']
 ['q5' 'b' 'q5']
 ['q5' 'a' 'q5']]

Alphabet: ['E' 'a' 'b']
States: ['q0' 'q1' 'q2' 'q3' 'q4' 'q5']
Initial State: q0
Final States: ['q5']
Transitions:
[['q0' 'E' 'q1']
 ['q0' 'E' 'q2']
 ['q1' 'b' 'q3']
 ['q2' 'a' 'q4']
 ['q3' 'E' 'q5']
 ['q4' 'E' 'q5']
 ['q5' 'b' 'q5']
 ['q5' 'a' 'q5']]
```

Figure 1

Diagrama de un NFA con 6 estados (q0 a q5). q0 es el estado inicial. q5 es el estado final. Las transiciones son: q0 a q1 y q2 por 'E'; q1 a q3 por 'b'; q2 a q4 por 'a'; q3 a q5 por 'E'; q4 a q5 por 'E'; q5 a q5 por 'a' y 'b'.

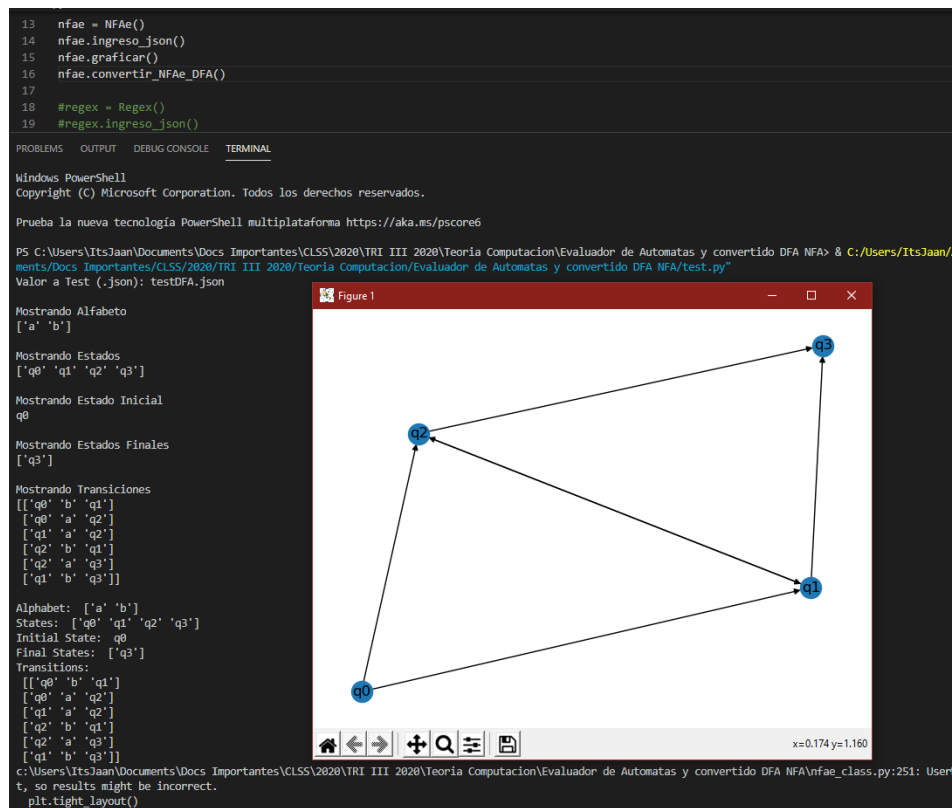
Convertido a DFA:



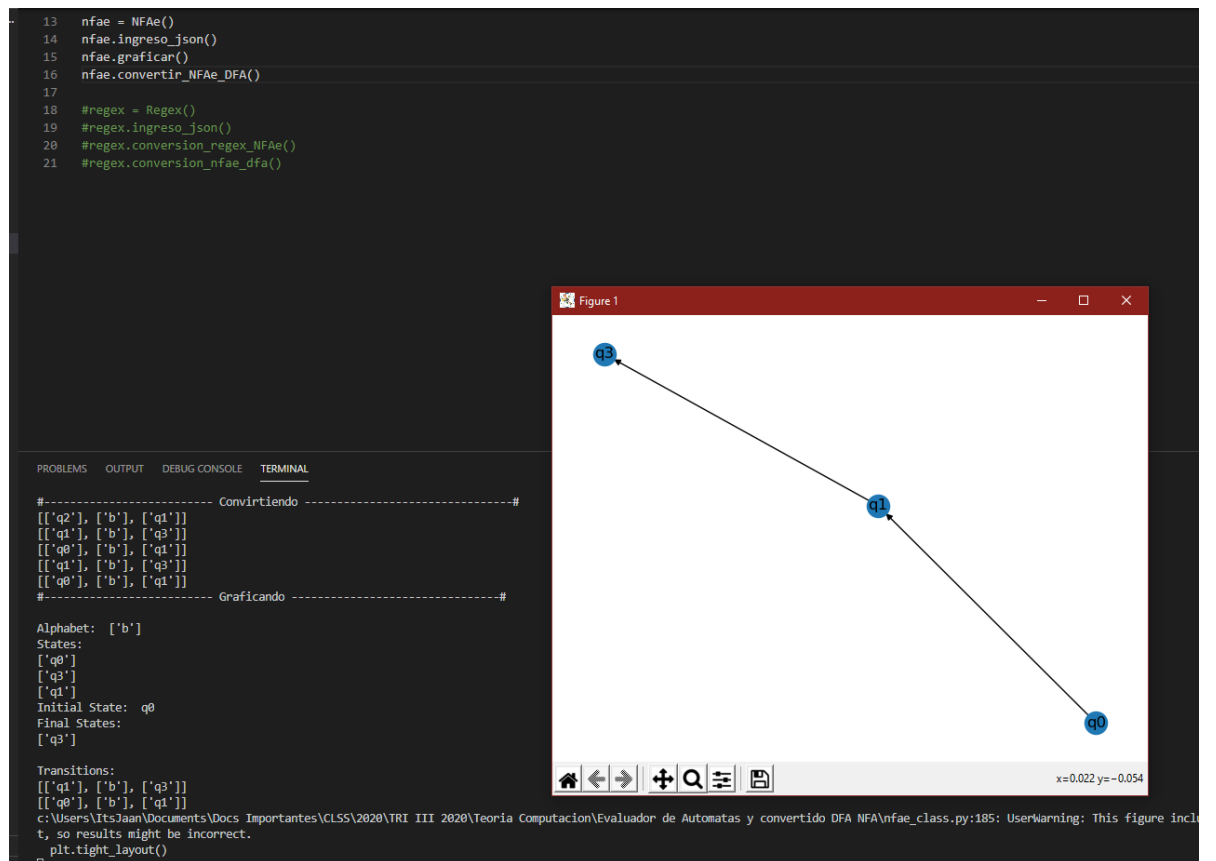
### Prueba 2:



## Resultado:



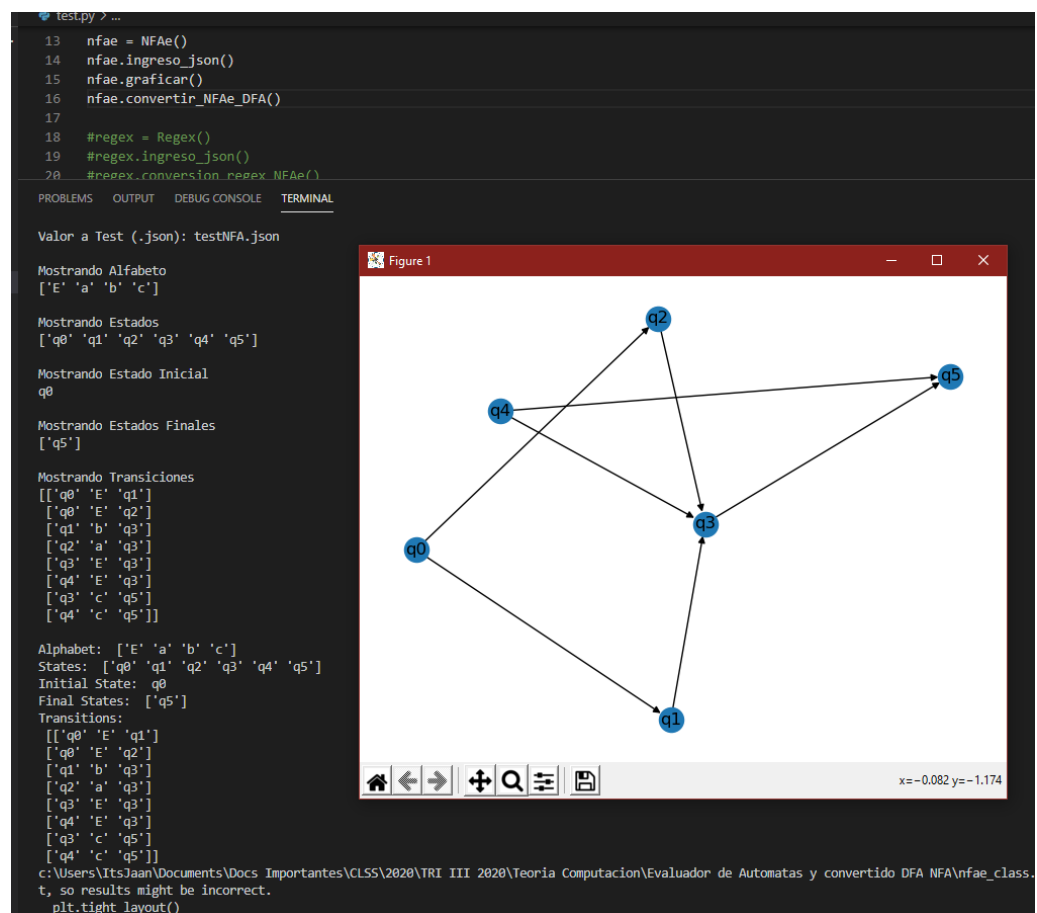
## Convertido a DFA:



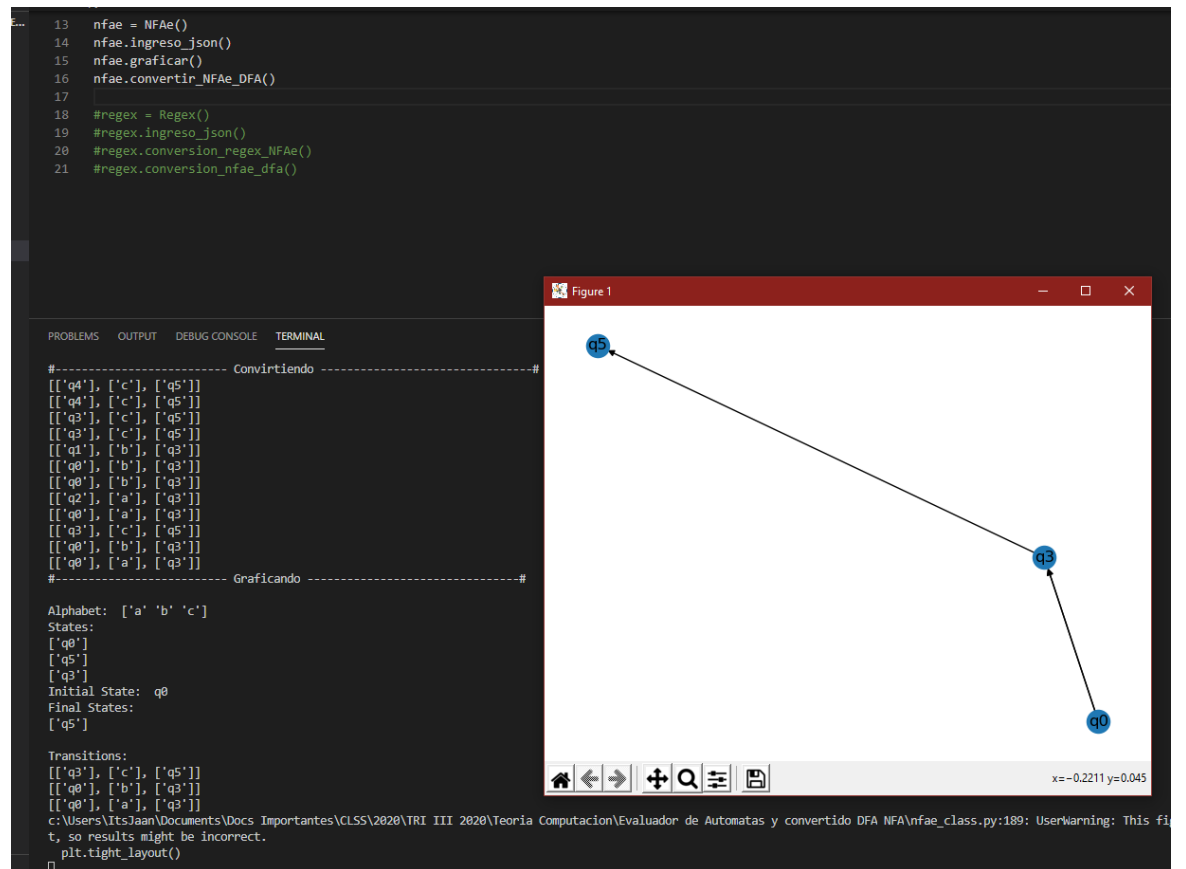
### Prueba 3:

```
test.py  testNFA.json X
testNFA.json > [ ] transitions
1  {
2    "alphabet": {
3      "E": "E",
4      "a": "a",
5      "b": "b",
6      "c": "c"
7    },
8    "states": {
9      "q0": "",
10     "q1": "",
11     "q2": "",
12     "q3": "",
13     "q4": "",
14     "q5": ""
15   },
16   },
17   "initial_state": "q0",
18   "accepting_states": {
19     "q5": ""
20   },
21   "transitions": [
22     [ "q0", "E", "q1"],
23     [ "q0", "E", "q2"],
24     [ "q1", "b", "q3"],
25     [ "q2", "a", "q3"],
26     [ "q3", "E", "q3"],
27     [ "q4", "E", "q3"],
28     [ "q3", "c", "q5"],
29     [ "q4", "c", "q5"]
30   ]
31 }
32 }
```

### Resultado:



Convertido a DFA:



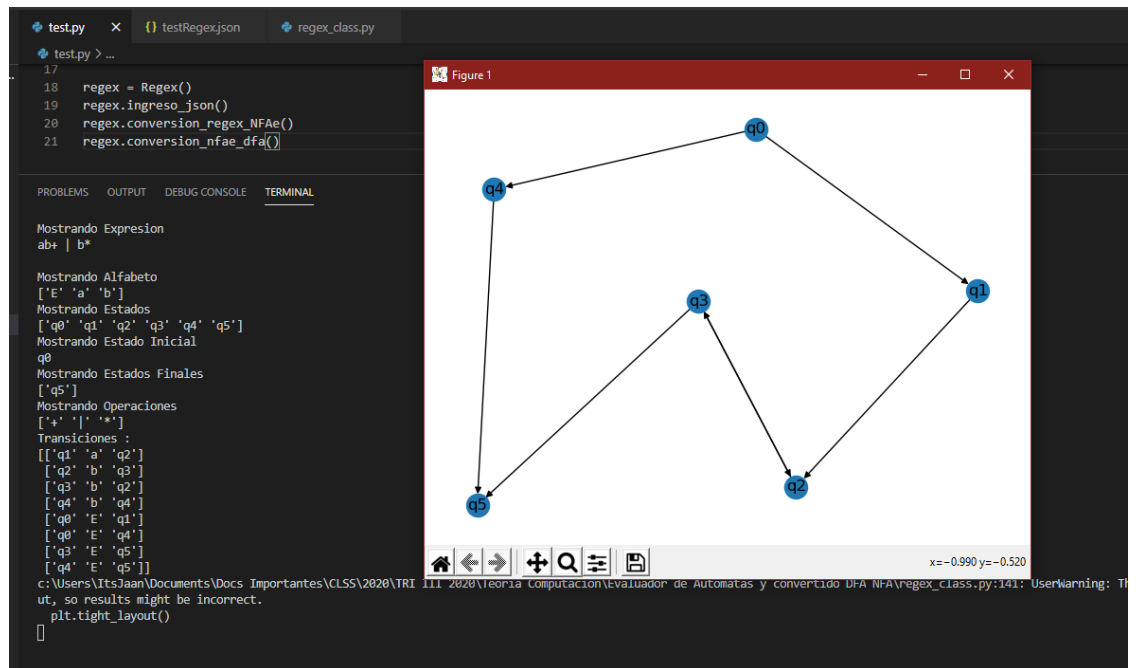
## 2.4 Prueba ER

### Prueba 1:

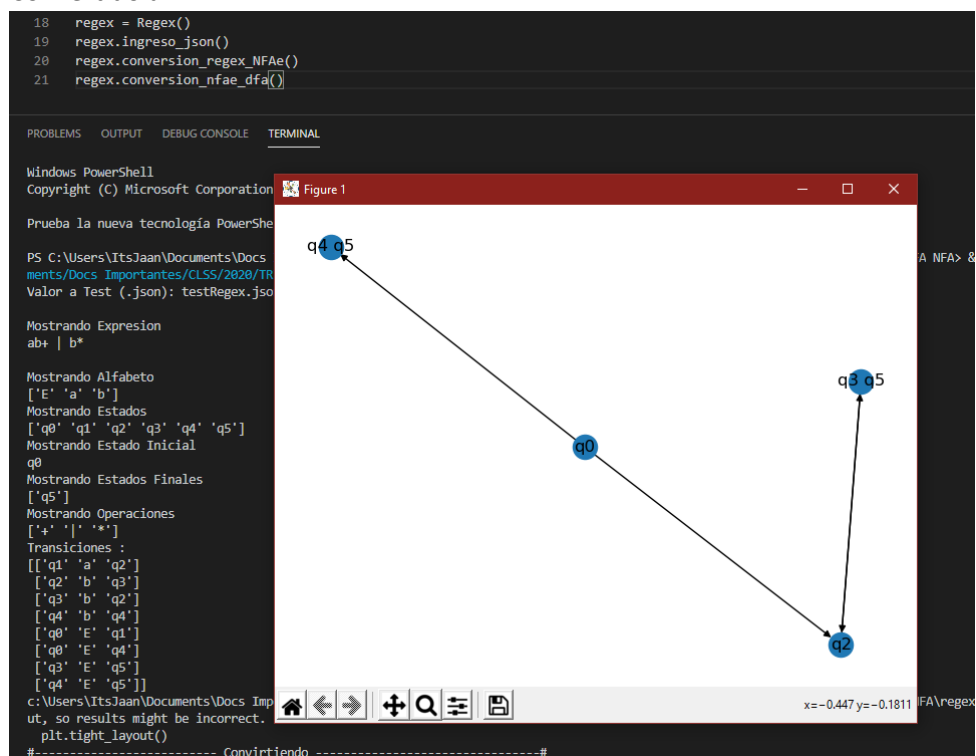
```
test.py  testRegex.json X  regex_class.py

testRegex.json > expression
1  {
2    "expression": "ab+ | b*"
3  }
```

### Resultado:



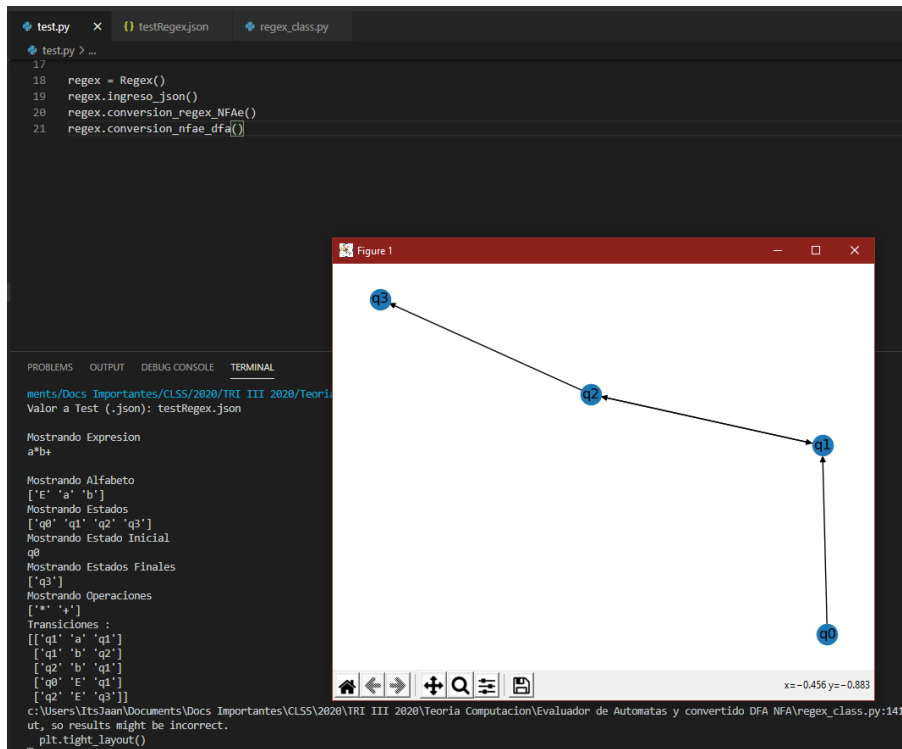
### Convertido a DFA:



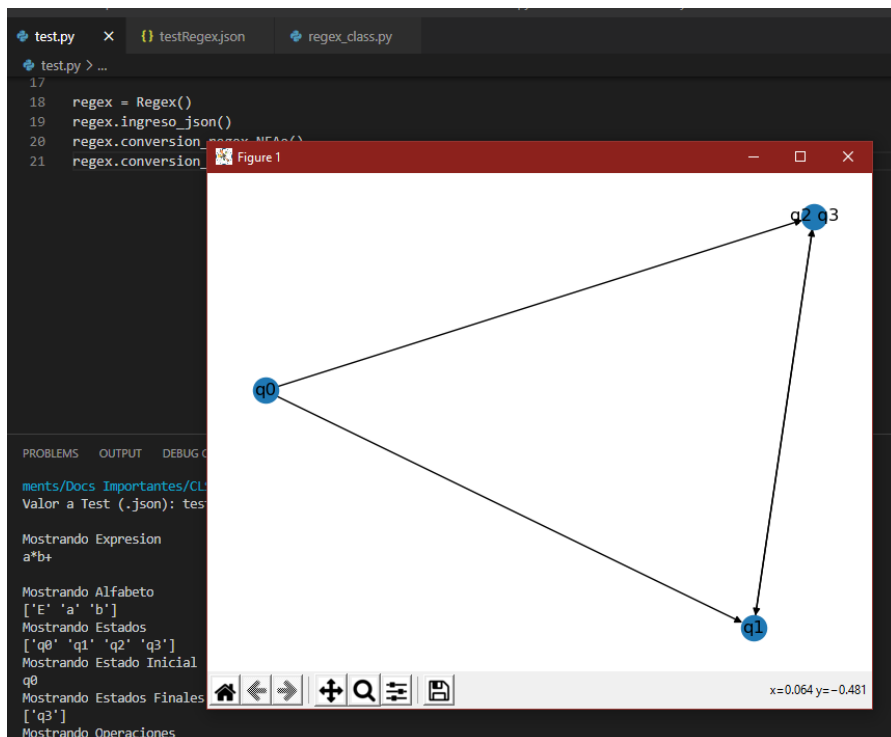
## Prueba 2:

```
test.py  testRegex.json  regex_class.py
testRegex.json > expression
1  {
2  "expresion": "a*b+"
3  }
```

## Resultado:



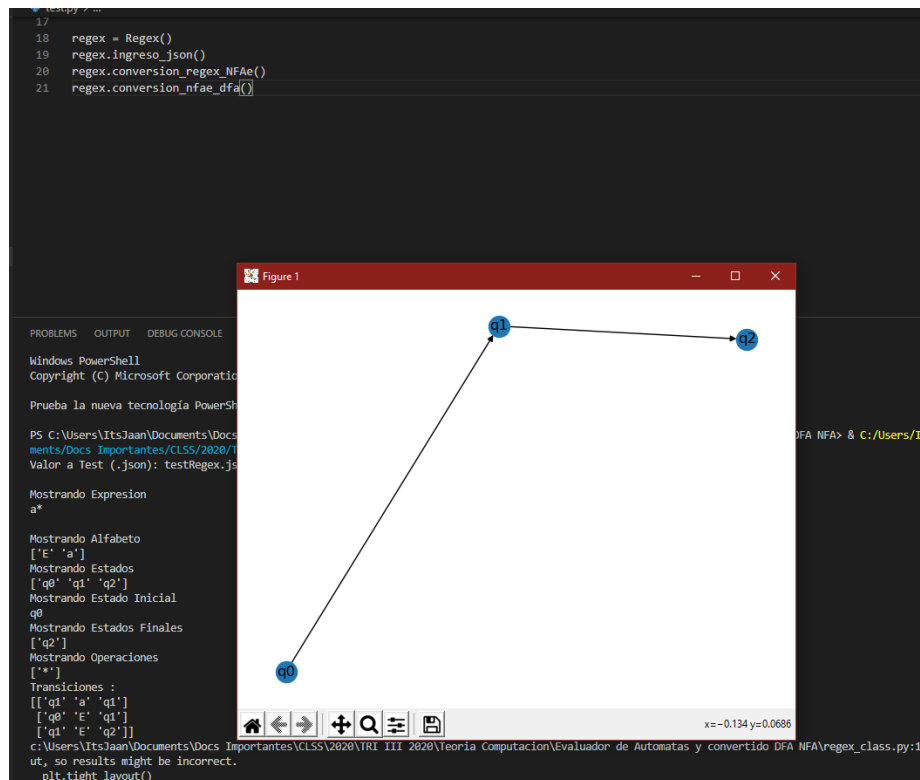
## Convertido a DFA:



### Prueba 3:

```
test.py  testRegexjson x  regex_class.py
testRegexjson > expression
1  {
2    "expresion": "a*"
3  }
```

### Resultado:



### Convertido a DFA:

