Clase:

Teoría de la Computación

Tarea:

Equivalencia DFA, NFA-E y ER

Catedrático:

Inge Cesar Orellana

Alumno:

21741236 Johnnie Miralda

Fecha:

San Pedro Sula, 31/08/2020

Tabla de contenido

Introducción:

En el proyecto propuesto para la clase de teoría de la computación se nos dio el reto de poner en práctica nuestros conocimientos de la teoría de autómata que es una rama de la teoría de la computación que estudia las maquinas abstractas y los problemas que éstas son capaces de resolver. Un autómata es un modelo matemático para una máquina de estado finito. Una máquina de estado finito es una máquina que, dada una entrada de símbolos, "salta" a través de una serie de estados de acuerdo con una función de transición.

El proyecto requiere que seamos capaces de manejar y encontrar sus equivalencias entre 4 tipos de autómatas siendo las Expresiones Regulares, Épsilon-NFA, NFA y DFA. En computo teórico y teoría de lenguajes formales una expresión regular es una secuencia de caracteres que conforman un patrono de búsqueda, se utilizan principalmente para la búsqueda de patrones de cadenas de caracteres u operaciones de sustitución. Un DFA (autómata finito determinista) es un que además es un sistema determinista, es decir que para cada estado en que se encuentre el autómata y con el alfabeto existe siempre no más de una transición posible desde ese estado con ese símbolo. Un NFA (autómata finito no determinista) es un autómata finito que a diferencia del determinista poseen almeno un estado que contiene más de una transición con el mismo elemento del alfabeto. Un E-NFA (autómata finito no determinista con transiciones vacías) es un NFA que contiene transiciones que permiten al autómata cambiar de estado sin procesar ningún símbolo de entrada, para hacer referencia a estas transiciones se les da el valor de ε(épsilon).

Para este proyecto el Ing. Cesar nos alentó a usar el lenguaje de programación Python, el cual gracias a su flexibilidad y facilidad fue de mucho provecho y nos ayudó a hacer este proyecto más fácil de hacer.

Composición de tecnologías utilizadas

Python:

El lenguaje de programación elegido para este proyecto fue Python, un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma. A continuación, hablaremos de los módulos que se utilizaron:

JSON lib:

Este módulo nos ayuda a poder leer archivos que están en sintaxis JSON y nos convierte la información en un diccionario de Python el cual usaremos para definir lo que son nuestros autómatas.

Networkx:

Es un paquete de Python para la creación, manipulación y el estudio de estructuras, dinámicas y funciones de redes complejas.

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. Proporciona una API, pylab, diseñada para recordar a la de MATLAB.

Json:

Es un acrónimo para JavaScript Objeto Notación, es un formato de texto sencillo para el intercambio de datos. Se trata de subconjuntos de la notación literal de objetos de JavaScript. Esta tecnología se utilizó de forma de entrada de información para cargar los autómatas al programa.

Implementación de proyecto

_init()

Para la implementación de este proyecto se hizo la creación de una clase llamada Autómata en la cual se desarrolla toda la lógica necesaria para leer el json, y con esa información transformar la y poder llegar a lo que es el DFA para poder evaluar la cadena que le metamos.

```
def __init__(self, link):
    json_file= open(link,"r")
    self.automata= json.load(json_file)
    json_file.close()
```

En la inicialización del autómata recibimos el nombre del archivo JSON del cual sacaremos la información para convertirla en un diccionario de Python.

dfa.json > ...

```
{
    "alphabeto":["0","1"],
    "estados":["A","B","C"],
    "e_inicial":"A",
    "e_final":["C"],
    "transiciones":[
        ["A","B","0"],
        ["B","C","1"],
        ["A","C","1"]
    ]
}
```

Podemos ver que el JSON nos dará la información de los alfabetos, estados, el estado inicial y los estados finales. En transiciones el orden de cómo se representan es la siguiente:

[estado actual, estado siguiente, valor de transición]

IS_ENFA

Toda la lógica del proyecto comienza llamando la función IS_ENFA () la cual verifica si la información recibida es de un E-NFA o NFA. Respectivamente después de haber identificado si es E_NFA se le agregan todas las transiciones propias con épsilon , consiguiente mente se manda a crear el archivo dos para poder mostrar la gráfica, después de eso se llama la función enfa_nfa() la cual convierte el diccionario de e-nfa a nfa, después de eso se vuelve a llamar la función draw() para poder sacar la gráfica NFA, después de eso se usa la función nfa_dfa() la cual convierte el NFA a NDA y por último se llama la función dfa_evaluar() la cual evalúa la cadena que se le ingresa adentro de ella.

```
212     def is_ENFA(self):
213         auto=self.automata
214         res= copy.deepcopy(auto)
215         if  "$" in res["alphabeto"]:
216             for x in range(len(res["estados"])):
217                 if x != "$":
218                     letra= res["estados"][x]
219                     res["transiciones"].append([letra,letra,"$"])
220 
221             self.draw("e-nfa.dot")
222             self.automata=res
223             self.enfa_nfa()
224 
225             self.draw("nfa.dot")
226             self.nfa_dfa()
227 
228             self.draw("dfa.dot")
229             self.dfa_evaluar()
230         else:
231             self.draw("nfa.dot")
232             self.nfa_dfa()
233 
234             self.draw("dfa.dot")
235             self.dfa_evaluar()
```

Enfa_nfa()

La lógica para pasar de E-NFA a NFA es primero que todo es hacer una copia del diccionario el cual lo ocuparemos para poder crear el nuevo diccionario NFA, de la copia del diccionario el alfabeto borrar la épsilon ($ en nuestro caso), después de la copia del diccionario se borran todas las transiciones que tengan que ver con épsilon. El siguiente algoritmo es para encontrar las transiciones con épsilon, primero tenemos un for para recorrer todo el alfabeto, el siguiente for es para encontrar por cada estado donde puede llegar, tr lo recorremos para encontrar todas las transiciones con nuestro estado principal que sean con épsilon, después de encontrar la transición con épsilon, se vuelve a recorrer las transiciones para encontrar la transición con el alfabeto del estado que llegamos con épsilon, después de eso volvemos a recorrer las transiciones para encontrar las transiciones con épsilon del estado encontrado

anteriormente para terminar el algoritmo se agrega la transición del estado principal al último estado encontrado con el valor del alfabeto que estamos iterando.

```
51          # declaramos las transiciones
52          for al in alfa:
53              for es in esta:
54                  for tr in tran:
55                      #comprueba epsilon del estado tr
56                      if tr[0] == es and tr[2]=="$":
57                          act= tr[1]
58                          for tr1 in tran:
59                              # comprueba si el epsilon de tr tiene conexion por al
60                              if tr1[0]==act and tr1[2]==al:
61                                  act2= tr1[1]
62                                  for tr2 in tran:
63                                      # comprieba si la conexion por al de tr1 tiene epsilon
64                                      if act2 == tr2[0] and tr2[2]=="$":
65                                          temp= [es,tr2[1],al]
66                                          if temp not in res["transiciones"]:
67                                              res["transiciones"].append(temp)
68
69
```

Nfa_dfa ()

En la función de pasar de NFA a DFA se hace lo mismo que en E-NFA, se clona el diccionario y se borran todas las transiciones, alfabeto, estados, y estados finales. Después de eso se inicia una lista con la respuesta de la función crearcombi (se definirá a continuación). En la lista tendremos el siguiente orden:

[estado, [transiciones con el primer valor del alfabeto, transición con el segundo valor, …]

En dado caso el estado no contiene una transición con el valor del alfabeto se guardará un @. A continuación, se crea un for para recorrer la lista anterior mente mencionada secundario se crea un for para recorrer la lista de transiciones del estado y revisar si están en el diccionario que estamos creando. Si no existen entran en un if en el cual se recorren todos los estados y s verifica si son parte de los estados, si son parte de los estados se hacer otro for para verificar si esta agregada al diccionario que estamos creando, si no está agregado se llama la función crearcombi () y su respuesta se agrega al diccionario que estamos creando. En dado caso no sea parte de los estados, esto es referente a que es una combinación de estados por lo tanto se llama la función crearcombi2(se definirá después) y su respuesta será agregada al diccionario que estamos creando. Podemos ver que el diccionario va creciendo mediante la lógica de agregar ya sea con creacombi o creacombi2. Después de esta lógica se quitan cualquier parte del diccionario que este duplicado en la función dup(). Después de eso vamos a recorrer la lista que creamos

anterior mente, y por cada elemento del diccionario y va a correr un for de la posición del alfabeto, y se crean las transiciones con el valor del estado que está en elem [0] con la posición del alfabeto en elem [1] [posición alfabeto] con el valor del alfabeto, cabe resaltar que se agrega también todos los estados mientras ya no estén agregados en estados. Para finalizar se recorren todos los estados y se identifican los que contengan el estado final anterior y se agregan a los estados finales del diccionario que vamos a devolver.

```
99          for x in dic:
100             for y in x[1]:
101                 val=True
102                 for p in dic:
103                     if y in p[0]:
104                         val=False
105 
106 
107                 if val:
108                     for z in x[1]:
109                         if z in estados:
110                             val2=True
111                             for c in dic:
112                                 if z in c[0]:
113                                     val2=False
114 
115                             if val2:
116                                 #print(z,"aqi")
117                                 dic.append(self.creacombi(z,auto))
118                         else:
119                             dic.append(self.creacombi2(z,auto))
120 
121         self.dup(dic)
```

Creacombi()

En crea combi la usamos para crear la lista que vamos a agregar en la lista de donde sacaremos todos las nuevas transiciones y estados. La función recibe la letra que ocupemos sus transiciones y el autómata como tal. Se crea una lista temporal para agregar estados, se recorren las transiciones y se encuentran todas las transiciones con la letra y el alfabeto que estemos probando en ese momento, eso se agrega a la lista, al terminar ese for la lista se hace un solo String y lo guardamos en la lista bajo la posición del alfabeto que se probó, en dado caso no se haya encontrado una transición en ese alfabeto se agregara un @ para recalcar que no hay una transición y así dando el respuesta que se manda de vuelta.

```
146         res=[[letra]]
147         res2=[]
148         for x in alfabeto:
149             combi=[]
150             prueba=True
151             for y in transiciones:
152                 if y[0] in letra and y[2]==x:
153                     combi+=y[1]
154                     prueba=False
155             if prueba:
156                 res2.append("@")
157             else:
158                 combi.sort()
159                 var= self.listToString(combi)
160                 res2.append(var)
```

Creacombi2()

La lógica de crea combi 2 es idéntica que la primera que ahora recibimos un String que contiene varios estados juntos. Se separan los estados y se crea un for para recorrer todos los estados y hacer la misma lógica que en creacombi solo que ahora sumando todos los alfabetos en la primera posición de todos los estados que recibimos en letra.

```
166     def creacombi2(self, letras, auto):
167             alfabeto= auto["alphabeto"]
168             transiciones= auto["transiciones"]
169             res=[[letras]]
170             letra= self.str_char(letras)
171             res2=[]
172             for x in alfabeto:
173                 combi=[]
174                 prueba=True
175                 for y in transiciones:
176                     if y[0] in letra and y[2]==x:
177                         if y[1] not in combi:
178                             combi+=y[1]
179                             prueba=False
180                 if prueba:
181                     res2.append("@")
182                 else:
183                     combi.sort()
184                     var= self.listToString(combi)
185                     res2.append(var)
186             res.append(res2)
187             return res
```

Draw()

La función draw se llama para poder crear un archivo tipo dot de cada uno de los tipos de autómatas, lo que recibe es el nombre de cómo se va a guardar el archivo. Lo primero que se hace es que se inicializa una variable DiGraph de la librería de networkx, después de eso recorremos las transiciones del autómata y verificamos que si los estados no están se agregan en modo de node al DiGraph, y consiguientemente se agregan las edges y el valor de transición se guarda a modo de label de ese Edge. Después de eso se crea la figura, y usando una función de networkx traducimos el DiGraph a un archivo en formato dot con el nombre que nos vino. Al tener el archivo dot usaremos la página http://www.webgraphviz.com/ para copiar el archivo dot y usar la facilidad de la pagina para mostrar la gráfica.

```
259         for x in trans:
260             temp= x[0] ,x[1] ,x[2]
261             label[(x[0], x[1])] = x[2]
262 
263             if x[0] not in cuales:
264                 cuales.append(x[0])
265                 G.add_node(x[0])
266 
267             if x[1] not in cuales:
268                 cuales.append(x[1])
269                 G.add_node(x[1])
270 
271             if len(edges)==0:
272                 edges.append(x)
273             else:
274                 paso=True
275                 for b in edges:
276                     if b[0]== x[0] and b[1]==x[1]:
277                         paso=False
278                         b[2]=b[2]+self.listToString(x[2])
279                 if paso:
280                     edges.append(x)
281 
282         for x in edges:
283             G.add_edge(x[0],x[1], label=x[2])
284 
```

Dfa_evaluar()

Esta función lo que hace es que rompe la cadena que recibimos de input y nos movemos a través de los nodos para ver si al final se queda en un acepting state y la cadena forma parte del lenguaje del autómata

```
264     def dfa_evaluar(self):
265         auto=self.automata
266         rep=True
267         while rep:
268             evaluar= input("Ingrese cadena a probar: ")
269             current_state = auto["e_inicial"]
270             transition_exists = True
271 
272             for char_index in range(len(evaluar)):
273                 current_char = evaluar[char_index]
274                 encontro= True
275                 for actual in range(len(auto["transiciones"])):
276                     actual_nodo= auto["transiciones"][actual]
277 
278                     if(current_state == actual_nodo[0] and current_char == actual_nodo[2]):
279                         current_state= actual_nodo[1]
280                         encontro=False
281                         break
282                 if encontro:
283                     transition_exists=False
284                     break;
285 
286             if current_state in auto["e_final"] and transition_exists:
287                     print ("Pertenece a L(M)")
288             else:
289                 print ("No pertenece a L(M)")
290 
291             res= input("Desea probar otra cadena? 1.Si 2.No ")
292             if res =="2":
293                 break
294
```

Tests

Test 1 E-nfa

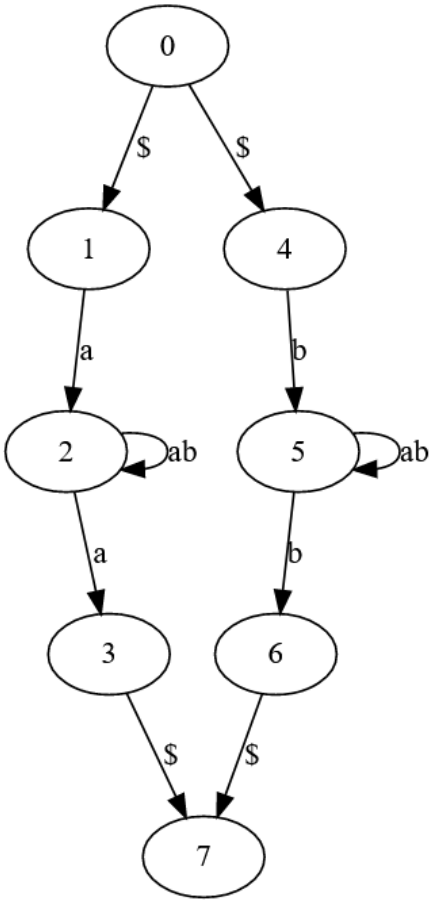
Esta es el autómata e-nfa que aremos en la prueba 1, sabemos que es un e-nfa ya que tiene épsilon representado por $ en su alfabeto y tiene transiciones con él.

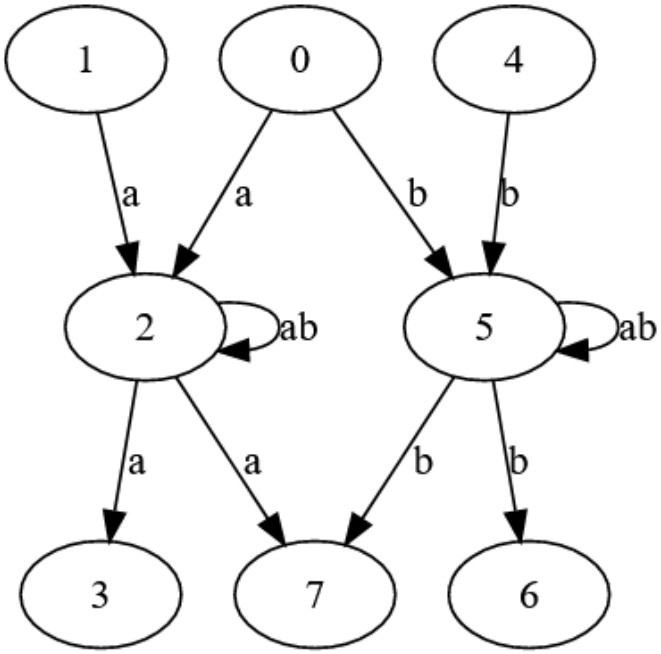
```
{
    "alphabeto":["a","b","$"],
    "estados":["0","1","2","3","4","5","6","7"],
    "e_inicial":"0",
    "e_final":["7"],
    "transiciones":[
        ["0","1","$"],
        ["0","4","$"],
        ["1","2","a"],
        ["2","2","a"],
        ["2","2","b"],
        ["2","3","a"],
        ["3","7","$"],
        ["4","5","b"],
        ["5","5","a"],
        ["5","5","b"],
        ["5","6","b"],
        ["6","7","$"]
    ]
}
```

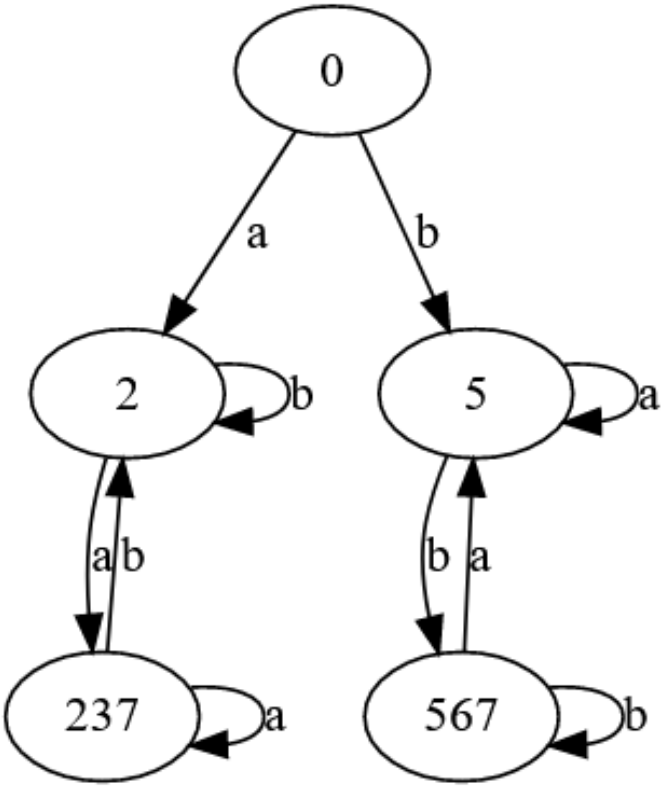
A continuación, podemos ver cada uno de los diccionarios en los dichos pasos de la conversión entre los e-nfa a nfa y por último el DFA.

```
Que archivo desea abrir: e-nfa.json
e-nfa.dot :
alfabeto ['a', 'b', '$']
estados: ['0', '1', '2', '3', '4', '5', '6', '7']
inicial: 0
finales: ['7']
transiciones: [['0', '1', '$'], ['0', '4', '$'], ['1', '2', 'a'], ['2', '2', 'a'], ['2', '2', 'b'], ['2', '3', 'a'], ['3', '7', '$'], ['4', '5', 'b'], ['5', '5', 'a'], ['5', '5', 'b'], ['5', '6', 'b'], ['6', '7', '$']]
_____
nfa.dot :
alfabeto ['a', 'b']
estados: ['0', '1', '2', '3', '4', '5', '6', '7']
inicial: 0
finales: ['7']
transiciones: [['1', '2', 'a'], ['2', '2', 'a'], ['2', '2', 'b'], ['2', '3', 'a'], ['4', '5', 'b'], ['5', '5', 'a'], ['5', '5', 'b'], ['5', '6', 'b'], ['0', '2', 'a'], ['2', '7', 'a'], ['0', '5', 'b'], ['5', '7', 'b']]
_____
dfa.dot :
alfabeto ['a', 'b']
estados: ['0', '2', '5', '237', '567']
inicial: 0
finales: ['237', '567']
transiciones: [['0', '2', 'a'], ['0', '5', 'b'], ['2', '237', 'a'], ['2', '2', 'b'], ['5', '5', 'a'], ['5', '567', 'b'], ['237', '237', 'a'], ['237', '2', 'b'], ['567', '5', 'a'], ['567', '567', 'b']]
```

A continuación, vemos las gráficas de cada uno de los autómatas.







Test 1_a

Para la primera prueba metimos una cadena pequeña de “aaa” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: aaa
Pertenece a L(M)
```

Test 1_b

Para la segunda prueba metimos una cadena más grande de “baaaabbbbb” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: baaaabbbbb
Pertenece a L(M)
```

Test 1_c

Para la tercera prueba metimos una cadena pequeña de “ababb” la que no debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: ababb
No pertenece a L(M)
```

Test 2 E-nfa

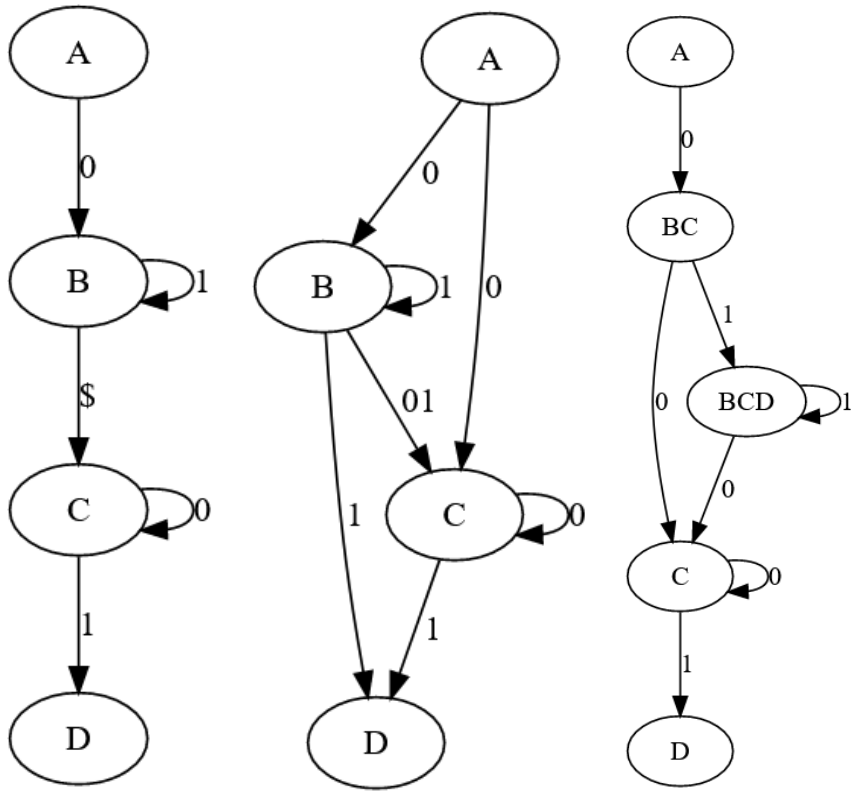
Esta es el autómata e-nfa que haremos en la prueba 2, sabemos que es un e-nfa ya que tiene épsilon representado por $ en su alfabeto y tiene transiciones con él.

```
{
    "alphabeto":["0","1","$"],
    "estados":["A","B","C","D"],
    "e_inicial":"A",
    "e_final":["D"],
    "transiciones":[
        ["A","B","0"],
        ["B","B","1"],
        ["B","C","$"],
        ["C","C","0"],
        ["C","D","1"]
    ]
}
```

A continuación, podemos ver cada uno de los diccionarios en los dichos pasos de la conversión entre los e-nfa a nfa y por último el DFA.

```
e-nfa.dot :
alfabeto ['0', '1', '$']
estados: ['A', 'B', 'C', 'D']
inicial: A
finales: ['D']
transiciones: [['A', 'B', '0'], ['B', 'B', '1'], ['B', 'C', '$'], ['C', 'C', '0'], ['C', 'D', '1']]
_________
nfa.dot :
alfabeto ['0', '1']
estados: ['A', 'B', 'C', 'D']
inicial: A
finales: ['D']
transiciones: [['A', 'B', '0'], ['B', 'B', '1'], ['C', 'C', '0'], ['C', 'D', '1'], ['A', 'C', '0'], ['B', 'C', '0'], ['B', 'D', '1'], ['B', 'C', '1']]
_________
dfa.dot :
alfabeto ['0', '1']
estados: ['A', 'BC', 'C', 'BCD', 'D']
inicial: A
finales: ['BCD', 'D']
transiciones: [['A', 'BC', '0'], ['BC', 'C', '0'], ['BC', 'BCD', '1'], ['C', 'C', '0'], ['C', 'D', '1'], ['BCD', 'C', '0'], ['BCD', 'BCD', '1']]
_________
```

A continuación, vemos las gráficas de cada uno de los autómatas.



Test 2_a

Para la primera prueba metimos una cadena pequeña de “01111” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: 01111
Pertenece a L(M)
```

Test 2_b

Para la segunda prueba metimos una cadena más grande de “0111110001” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: 0111110001
Pertenece a L(M)
```

Test 2_c

Para la tercera prueba metimos una cadena pequeña de “010000” la que no debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: 010000
No pertenece a L(M)
```

Test 3 E-nfa

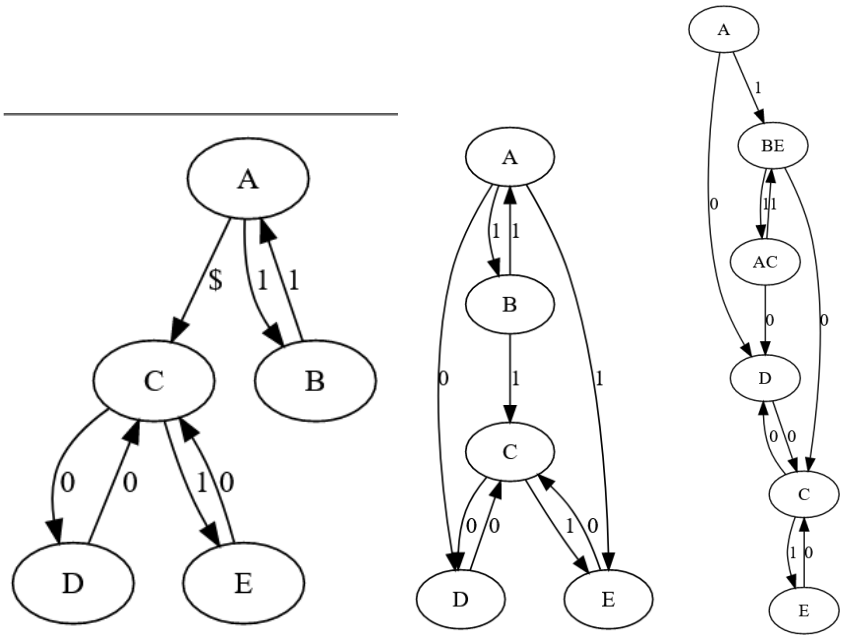
Esta es el autómata e-nfa que haremos en la prueba 3, sabemos que es un e-nfa ya que tiene épsilon representado por $ en su alfabeto y tiene transiciones con él.

```
{
    "alphabeto":["0","1","$"],
    "estados":["A","B","C","D","E"],
    "e_inicial":"A",
    "e_final":["C"],
    "transiciones":[
        ["A","C","$"],
        ["A","B","1"],
        ["B","A","1"],
        ["C","D","0"],
        ["C","E","1"],
        ["D","C","0"],
        ["E","C","0"]
    ]
}
```

A continuación, podemos ver cada uno de los diccionarios en los dichos pasos de la conversión entre los e-nfa a nfa y por último el DFA.

```
e-nfa.dot :
alfabeto ['0', '1', '$']
estados: ['A', 'B', 'C', 'D', 'E']
inicial: A
finales: ['C']
transiciones: [['A', 'C', '$'], ['A', 'B', '1'], ['B', 'A', '1'], ['C', 'D', '0'], ['C', 'E', '1'], ['D', 'C', '0'], ['E', 'C', '0']]
_____
nfa.dot :
alfabeto ['0', '1']
estados: ['A', 'B', 'C', 'D', 'E']
inicial: A
finales: ['C']
transiciones: [['A', 'B', '1'], ['B', 'A', '1'], ['C', 'D', '0'], ['C', 'E', '1'], ['D', 'C', '0'], ['E', 'C', '0'], ['A', 'D', '0'], ['A', 'E', '1'], ['B', 'C', '1']]
_____
dfa.dot :
alfabeto ['0', '1']
estados: ['A', 'D', 'BE', 'C', 'AC', 'E']
inicial: A
finales: ['C', 'AC']
transiciones: [['A', 'D', '0'], ['A', 'BE', '1'], ['D', 'C', '0'], ['BE', 'C', '0'], ['BE', 'AC', '1'], ['C', 'D', '0'], ['C', 'E', '1'], ['AC', 'D', '0'], ['AC', 'BE', '1'], ['E', 'C', '0']]
```

A continuación, vemos las gráficas de cada uno de los autómatas.



Test 3_a

Para la primera prueba metimos una cadena pequeña de “10” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: 10
Pertenece a L(M)
```

Test 3_b

Para la segunda prueba metimos una cadena más grande de “110010” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: 110010
Pertenece a L(M)
```

Test 3_c

Para la tercera prueba metimos una cadena pequeña de “00001” la que no debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: 00001
No pertenece a L(M)
```

Test 4 E-nfa

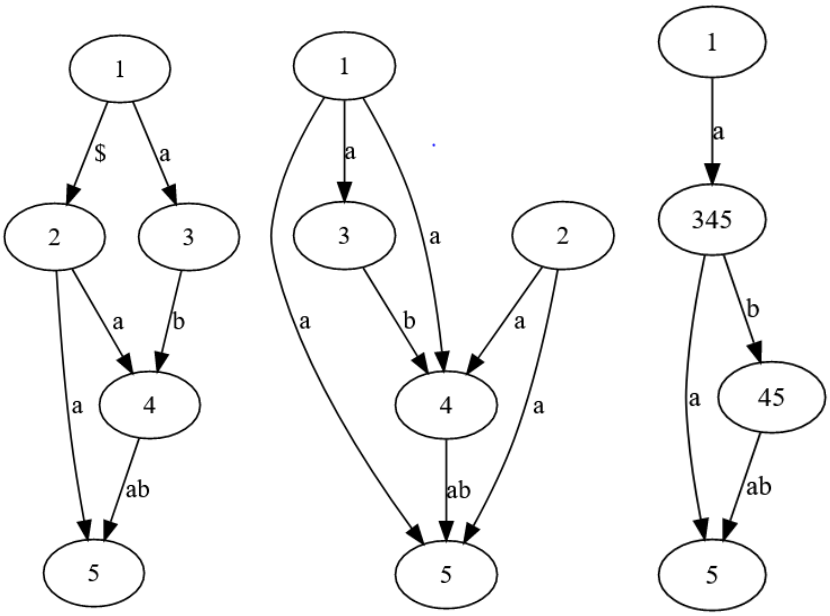
Esta es el autómata e-nfa que haremos en la prueba 4, sabemos que es un e-nfa ya que tiene épsilon representado por $ en su alfabeto y tiene transiciones con él.

```
{
    "alphabeto":["a","b","$"],
    "estados":["1","2","3","4","5"],
    "e_inicial":"1",
    "e_final":["5"],
    "transiciones":[
        ["1","2","$"],
        ["1","3","a"],
        ["2","5","a"],
        ["2","4","a"],
        ["3","4","b"],
        ["4","5","a"],
        ["4","5","b"]
    ]
}
```

A continuación, podemos ver cada uno de los diccionarios en los dichos pasos de la conversión entre los e-nfa a nfa y por último el DFA.

```
Que archivo desea abrir: e-nfa4.json
e-nfa.dot :
alfabeto ['a', 'b', '$']
estados: ['1', '2', '3', '4', '5']
inicial: 1
finales: ['5']
transiciones: [['1', '2', '$'], ['1', '3', 'a'], ['2', '5', 'a'], ['2', '4', 'a'], ['3', '4', 'b'], ['4', '5', 'a'], ['4', '5', 'b']]
_____
nfa.dot :
alfabeto ['a', 'b']
estados: ['1', '2', '3', '4', '5']
inicial: 1
finales: ['5']
transiciones: [['1', '3', 'a'], ['2', '5', 'a'], ['2', '4', 'a'], ['3', '4', 'b'], ['4', '5', 'a'], ['4', '5', 'b'], ['1', '5', 'a'], ['1', '4', 'a']]
_____
dfa.dot :
alfabeto ['a', 'b']
estados: ['1', '345', '5', '45']
inicial: 1
finales: ['345', '5', '45']
transiciones: [['1', '345', 'a'], ['345', '5', 'a'], ['345', '45', 'b'], ['45', '5', 'a'], ['45', '5', 'b']]
_____
```

A continuación, vemos las gráficas de cada uno de los autómatas.



Test 4_a

Para la primera prueba metimos una cadena pequeña de “a” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: a
Pertenece a L(M)
```

Test 4_b

Para la segunda prueba metimos una cadena más grande de “abb” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: abb
Pertenece a L(M)
```

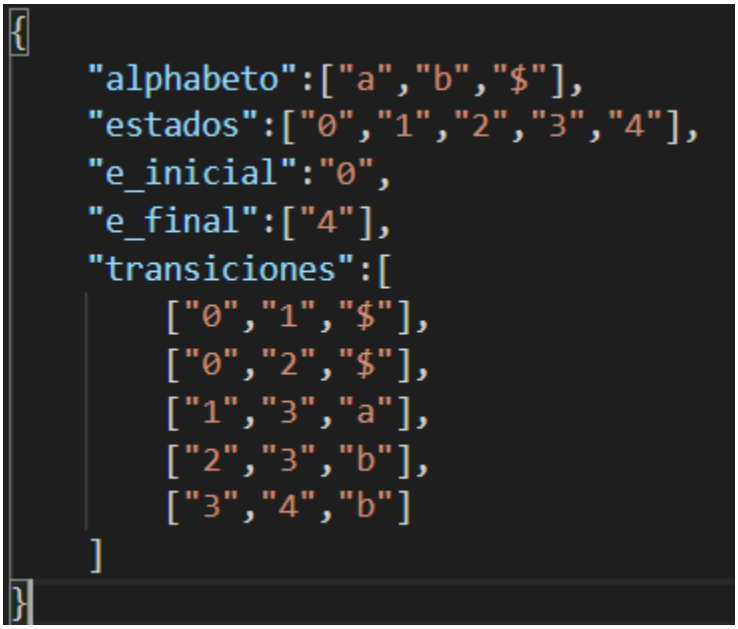
Test 4_c

Para la tercera prueba metimos una cadena pequeña de “baaa” la que no debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: baaa
No pertenece a L(M)
```

Test 5 E-nfa

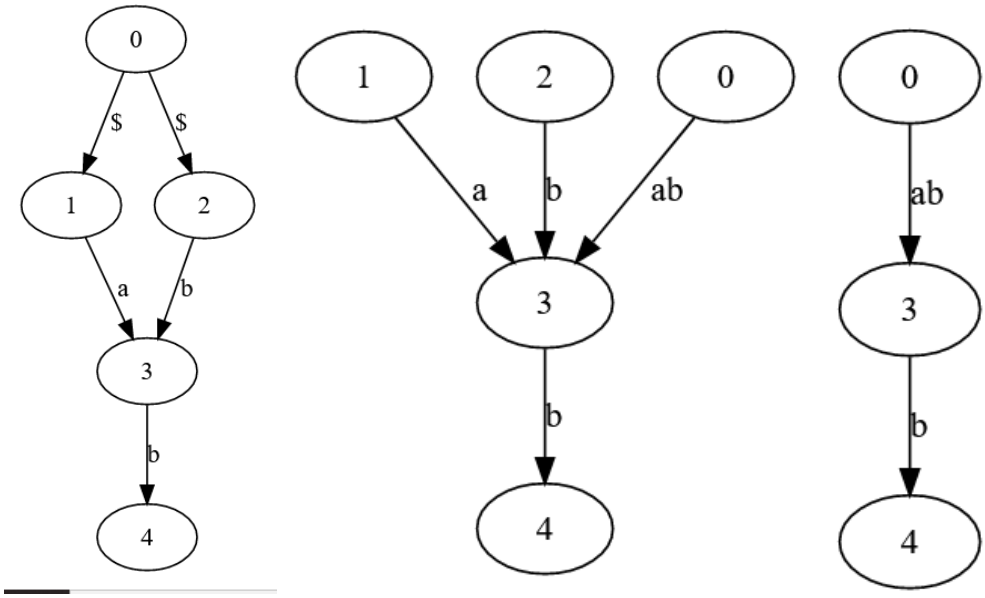
Esta es el autómata e-nfa que haremos en la prueba 5, sabemos que es un e-nfa ya que tiene épsilon representado por $ en su alfabeto y tiene transiciones con él.



A continuación, podemos ver cada uno de los diccionarios en los dichos pasos de la conversión entre los e-nfa a nfa y por último el DFA.

```
e-nfa.dot :
alfabeto ['a', 'b', '$']
estados: ['0', '1', '2', '3', '4']
inicial: 0
finales: ['4']
transiciones: [['0', '1', '$'], ['0', '2', '$'], ['1', '3', 'a'], ['2', '3', 'b'], ['3', '4', 'b']]
_________
nfa.dot :
alfabeto ['a', 'b']
estados: ['0', '1', '2', '3', '4']
inicial: 0
finales: ['4']
transiciones: [['1', '3', 'a'], ['2', '3', 'b'], ['3', '4', 'b'], ['0', '3', 'a'], ['0', '3', 'b']]
_________
dfa.dot :
alfabeto ['a', 'b']
estados: ['0', '3', '4']
inicial: 0
finales: ['4']
transiciones: [['0', '3', 'a'], ['0', '3', 'b'], ['3', '4', 'b']]
```

A continuación, vemos las gráficas de cada uno de los autómatas.



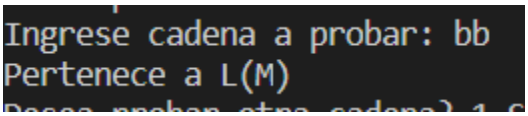
Test 5_a

Para la primera prueba metimos una cadena pequeña de “ab” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: ab
Pertenece a L(M)
```

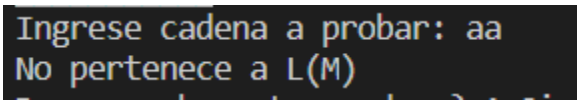
Test 5_b

Para la segunda prueba metimos una cadena más grande de “bb” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.



Test 5_c

Para la tercera prueba metimos una cadena pequeña de “aa” la que no debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.



Test 6 E-nfa

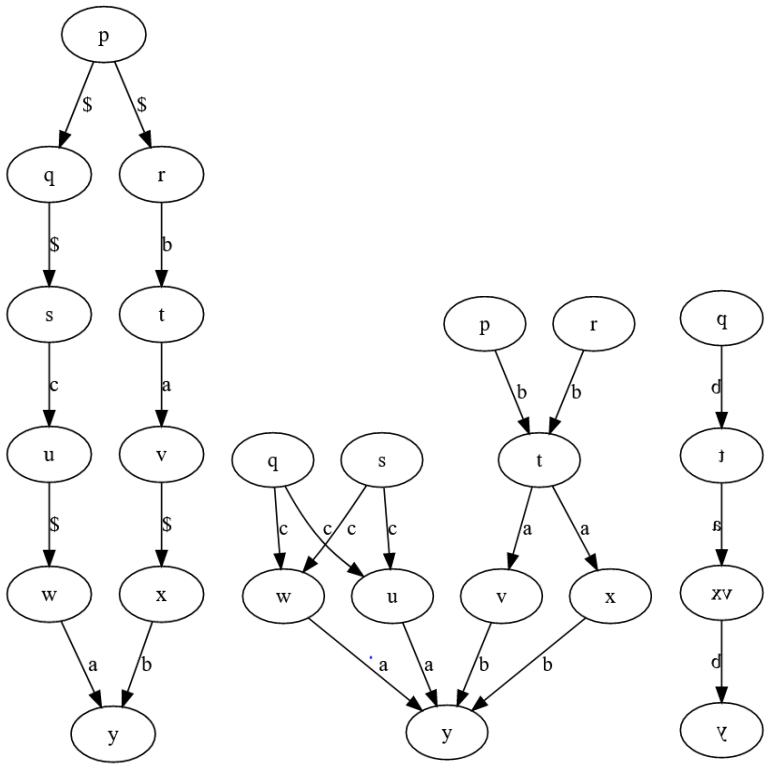
Esta es el autómata e-nfa que haremos en la prueba 6, sabemos que es un e-nfa ya que tiene épsilon representado por $ en su alfabeto y tiene transiciones con él.

```
{
    "alphabeto":["a","b","c","$"],
    "estados":["p","q","r","s","t","u","v","w","x","y"],
    "e_inicial":"p",
    "e_final":["y"],
    "transiciones":[
        ["p","q","$"],
        ["p","r","$"],
        ["q","s","$"],
        ["r","t","b"],
        ["s","u","c"],
        ["t","v","a"],
        ["u","w","$"],
        ["v","x","$"],
        ["w","y","a"],
        ["x","y","b"]
    ]
}
```

A continuación, podemos ver cada uno de los diccionarios en los dichos pasos de la conversión entre los e-nfa a nfa y por último el DFA.

```
e-nfa.dot :
alfabeto ['a', 'b', 'c', '$']
estados: ['p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y']
inicial: p
finales: ['y']
transiciones: [['p', 'q', '$'], ['p', 'r', '$'], ['q', 's', '$'], ['r', 't', 'b'], ['s', 'u', 'c
'], ['t', 'v', 'a'], ['u', 'w', '$'], ['v', 'x', '$'], ['w', 'y', 'a'], ['x', 'y', 'b']]
_____
nfa.dot :
alfabeto ['a', 'b', 'c']
estados: ['p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y']
inicial: p
finales: ['y']
transiciones: [['r', 't', 'b'], ['s', 'u', 'c'], ['t', 'v', 'a'], ['w', 'y', 'a'], ['x', 'y', 'b'], ['t', 'x', 'a'], ['u', 'y', 'a'], ['p', 't', 'b'], ['v', 'y', 'b'], ['q', 'w', 'c'], ['q', 'u', 'c'],
 ['s', 'w', 'c']]
_____
dfa.dot :
alfabeto ['a', 'b', 'c']
estados: ['p', 't', 'vx', 'y']
inicial: p
finales: ['y']
transiciones: [['p', 't', 'b'], ['t', 'vx', 'a'], ['vx', 'y', 'b']]
```

A continuación, vemos las gráficas de cada uno de los autómatas.



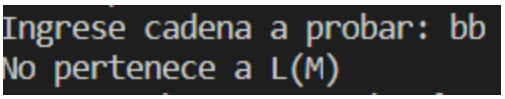
Test 6_a

Para la primera prueba metimos una cadena pequeña de “bab” la que debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.

```
Ingrese cadena a probar: bab
Pertenece a L(M)
```

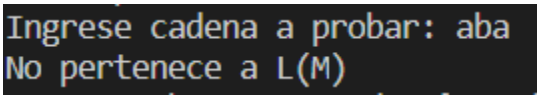
Test 6_b

Para la segunda prueba metimos una cadena más grande de “bb” la que no debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.



Test 6_c

Para la tercera prueba metimos una cadena pequeña de “aba” la que no debería ser aceptada, y efectivamente nos da la respuesta que buscábamos.



Bibliografía

Announcement: NetworkX 2.3 — NetworkX 2.5 documentation. (s. f.). Recuperado de https://networkx.github.io/documentation/stable/release/release_2.3.html

colaboradores de Wikipedia. (s. f.-a). Autómata finito determinista. Recuperado de https://es.wikipedia.org/wiki/Aut%C3%B3mata_finito_determinista

colaboradores de Wikipedia. (s. f.-b). Expresión regular. Recuperado de https://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

colaboradores de Wikipedia. (s. f.-c). Teoría de autómatas. Recuperado de https://es.wikipedia.org/wiki/Teor%C3%ADa_de_aut%C3%B3matas

colaboradores de Wikipedia. (2020a, abril 27). Matplotlib. Recuperado de https://es.wikipedia.org/wiki/Matplotlib

colaboradores de Wikipedia. (2020b, agosto 14). Autómata finito no determinista. Recuperado de https://es.wikipedia.org/wiki/Aut%C3%B3mata_finito_no_determinista

colaboradores de Wikipedia. (2020c, agosto 25). JSON. Recuperado de https://es.wikipedia.org/wiki/JSON

colaboradores de Wikipedia. (2020d, agosto 26). Python. Recuperado de https://es.wikipedia.org/wiki/Python

Conversion of Epsilon-NFA to NFA. (s. f.). Recuperado de https://www.geeksforgeeks.org/conversion-of-epsilon-nfa-to-nfa/

Python JSON. (s. f.). Recuperado de https://www.w3schools.com/python/python_json.asp

Wikipedia contributors. (s. f.). Nondeterministic finite automaton. Recuperado de https://en.wikipedia.org/wiki/Nondeterministic_finite_automaton