INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

GARZÓN DOMÍNGUEZ GERARDO ISMAEL

3CV18

COMPILADORES

PRÁCTICA 2:

AFN-e a AFD, autómata finito no determinista con transiciones epsilon a autómata finito determinista.

INTRODUCCIÓN

A continuación se presentan los conceptos clave para el análisis y desarrollo de esta práctica.

Es un hecho que los autómatas finitos no deterministas (AFN) pueden ser descritos por autómatas finitos deterministas (AFD), en la práctica, un AFD tiene al menos la misma cantidad de estados que un AFN, aunque con frecuencia, los AFD's tendrán más estados, en el peor caso un AFD tendrá al menos 2^n estados, mientras que un AFN tendrá n estados. [1]

El objetivo de convertir un AFN N a un AFD D, tal que $L(N) = L(D)$, esto es, que el lenguaje que acepta el autómata finito no determinista sea el mismo lenguaje que acepta el autómata finito determinista, esto se realizará mediante construcción de subconjuntos. [1]

La idea general detrás de este método, en pocas palabras, es que, un estado de un AFD corresponde a un conjunto de estados de un AFN [2]

El objetivo será construir una tabla de transiciones para el autómata D, y como ya se mencionó, cada estado del autómata D corresponde a un conjunto de estados del autómata N, de tal manera que cada transición que se realiza en D, simula una serie de transiciones en N en "paralelo", o sea en vez de hacerlas secuencialmente, se realizan "a la vez" en el AFD. [2]

Dos de los principales problemas para generar un AFD a partir de la definición de un AFN, son:

- Encontrar la equivalencia del estado inicial del AFN en el AFD, esto debido a que las transiciones ϵ están permitidas, por extensión, el autómata puede encontrarse inicialmente en cualquier estado que contenga transiciones ϵ. [2] Para resolver este problema se recurre a la definición de una función llamada $eclose$, (cerradura epsilon o *epsilon closure*), que se explicará más adelante.
- Encontrar la equivalencia del estado final del AFN en el AFD, en este caso, el conjunto de estados finales serán los subconjuntos generados a partir de los estados de N que contengan al menos un estado de aceptación del AFN N. [1]

Las siguientes operaciones son necesarias para proceder con este método:

- $eclose(T)$: Conjunto de estados de un AFN alcanzables mediante transiciones ϵ, para todo estado en T; $s \in T$ incluyendo el estado s. [2]
- $move(T, \alpha)$: Conjunto de estados de un AFN para los cuales existe una transición con el símbolo de entrada α, para cada estado $s \in T$. [2]
- Esta función no se encuentra explícitamente en la bibliografía, pero se hace su uso en la descripción un algoritmo[2], la llamamos y definimos como $go\_to(T, \alpha) = eclose(move(T, \alpha))$.

El problema de como identificar el estado inicial del AFN N en el nuevo AFD D se resuelve usando la función $eclose$, de tal manera que el conjunto generado por esta con el estado inicial del autómata N es el estado inicial del autómata D, esto es $A = q_{d0} = eclose(q_{n0})$. Para generar el nuevo autómata D, se comenzará aplicando la función la función $goto$ con cada estado nuevo que se genere y cada símbolo del alfabeto del autómata N, esto mientras el resultado de dicha función no se repita dentro de los subconjuntos ya generados, de lo contrario se continua aplicando sucesivamente:
$B = go\_to(A, \Sigma) = eclose(move(A, \Sigma)), C = go\_to(B, \Sigma), ...$

DESARROLLO E IMPLEMENTACIÓN

En esta sección se presentará la implementación para esta práctica.

En este caso se reutilizará cierta porción del código de la práctica anterior, en especial las funcionalidades de la clase $Automata$ para escanear y generar un autómata a partir de una definición en un archivo de texto, ya que a partir del objeto generado será posible auxiliarse de sus atributos, como el alfabeto, estados y sy método $Delta$, el cual permite mapear una llave o par ordenado $(q \in Q, \alpha \in \Sigma)$ a un conjunto de estados del AFN. Además, con la implementación de los nuevos métodos mostrados en la sección introductoria será posible convertir un AFN definido en un archivo de texto a un AFD y dar su definición en algún descriptor de salida, como la terminal o un archivo.

Para esta práctica se creó una nueva clase: $dfagen$, (*deterministic finite automata generator* o generador de autómata finito determinista). Como ya se mencionó, esta clase se auxiliará fuertemente de algunos atributos y funcionalidades de la clase $Automata$, de hecho, todo objeto $dfagen$ tiene asociado un objeto $Automata$, y el constructor para los objetos $dfagen$ recibe una ruta de archivo para crear un autómata.

En la Figura 1 se muestra el constructor por defecto de la clase $dfagen$, por ahora la funcionalidad final que se le da, es mostrar la definición del nuevo AFD en la terminal en el mismo formato del que se construye el AFN, esto es:

[linea 1] <conjunto de estados Q del autómata separados por comas>
[linea 2] <alfabeto del autómata Σ, elementos separados por comas>
[linea 3] <estado inicial del autómata $q_0 \in Q$>
[linea 4] <estados de aceptación del autómata $F \subseteq Q$, separados por comas>
[linea 5 …] <definición de la función de transiciones a_i, b_i, c_i> $i = 5, ...n$, donde n es el último renglón no vacío del archivo.

```
def __init__(self, filepath:str):
    self.NFA = automaton.Automata(filepath, ',')
    self.__finite_map__ = {}
    self.__sets_as_state__ = {}
    self.__stateno__ = 0
    self.__initial_state__ = frozenset(
        self.__ECLOSE__({self.NFA.__initial_state__})
    )

    self.DFA_from_DFN(self.__initial_state__)

    print(self.__format__())
```

Figura 1: Constructor por defecto de la clase $dfagen$

A continuación se presenta información sobre cada atributo de los objetos de tipo $dfagen$
$\_\_finite\_map\_\_$: Un diccionario que contiene la definición de la nueva función δ para el AFD.

$\_\_sets\_as\_state\_\_$: Un diccionario que mapea algún conjunto de estados generados durante la ejecución el algoritmo pertenecientes al AFN, a una representación de estado para el AFD, en este caso se eligieron números para representar los estados del AFD.

$\_\_state\_no\_\_$: Numeración de los estados generados durante la ejecución del algoritmo, esto es $0, 1, 2, ..., N-1$, donde N es la cantidad de estados en AFD equivalente.

$\_\_initial\_state\_\_$: El estado inicial del AFD.

A continuación se muestra la implementación de la operación $eclose$ y se ofrece una descripción de su funcionamiento.

```
def __ECLOSE_R__(self, Eset:set, state):
    key = (state, automaton.epsilon)
    S = self.NFA.Delta(key)
    if S[0] != None:
        for s in S:
            self.__ECLOSE_R__(Eset, s)
        Eset.update(S)

def __ECLOSE__(self, states:set):
    E = set()
    E.update(states)

    for e in states:
        self.__ECLOSE_R__(E, e)

    return E
```

Figura 2: Implementación de la operación $eclose$ mediante 2 funciones

La función $\_\_ECLOSE\_\_$ se encarga de crear un conjunto que incluya los estados de los cuales se computará su cerradura epsilon, y usará la función recursiva $\_\_ECLOSE\_R\_\_$ para encontrar todas las transiciones vacías que se pueden hacer desde los elementos de la variable $states$ que recibe como argumento. La $\_\_ECLOSE\_R\_\_$ se encarga de computar las transiciones vacías de manera recursiva para cada elemento que recibe como argumento, nótese como se hace uso de la función de transiciones del autómata asociado NFA.

```
def __MOVE__(self, states:set, symbol:str):
    M = set()

    for s in states:
        key = (s, symbol)
        D = self.NFA.Delta(key)
        if D[0] != None:
            M.update(D)

    return M
```

Figura 3: Implementación de la operación $move$

La siguiente función a implementar es para la operación $move$, en se muestra la implementación usada, cabe decir que esta función es menos compleja que aquellas para la operación $eclose$, ya que esta vez no es necesario buscar las transiciones de manera recursiva para todo subconjunto generado, en sí bastará con consultar la función δ del AFN para cada estado del conjunto y símbolo que $move$ recibe como argumentos.

Como última explicación respecto a la implementación se muestra como la función DFA_from_DFN en la Figura 4, hace uso conjunto de los atributos y funcionalidades de $dfagen$ para generar la información necesaria para definir el nuevo AFD.

```
def DFA_from_DFN(self, states:set):
    for a in self.NFA.__alphabet__:
        r = self.__ECLOSE__( self.__MOVE__(states, a) )
        fr = frozenset(r)
        fs = frozenset(states)

        self.__add_set__(fs)
        self.__add_set__(fr)

        # print(states, self.__sets_as_state__[fs], r, self.__se
        k = (self.__sets_as_state__[fs], a)
        if k not in self.__finite_map__:
            self.__finite_map__[k] = self.__sets_as_state__[fr]
            self.DFA_from_DFN(r)
```

Figura 4: Función para generar la información para definir el AFD equivalente al AFN

La función DFA_from_DFN es recursiva, y como se puede observar, esta se detiene cuando se generá alguna llave ya existente en el diccionario $\_\_finite\_map\_\_$, observe que la tercera linea de la función hace uso de la operación $goto$ definida previamente. Existen también ciertos detalles de implementación, como convertir los conjuntos a “conjuntos congelados”, esto debido a que *python* no permite *hashing* de objetos mutables [3].
Si bien se hace uso de una función recursiva, la solución iterativa también existe y es muy similar a la propuesta aquí [2].

A continuación se mostrarán ejemplos del funcionamiento de este programa con 2 entradas.

Para el primer autómata de prueba cuyo grafo se muestra en la Figura 5 en la cuya tabla de transiciones es la Tabla 1.

Tabla 1: Tabla de transiciones del autómata de prueba 1

Q	a	b	ϵ
→ 0	ϕ	ϕ	{1,8}
1	ϕ	ϕ	{2,7}
2	ϕ	ϕ	{3,5}
3	ϕ	{4}	ϕ
4	ϕ	ϕ	{5,3}
5	{6}	ϕ	ϕ
6	ϕ	ϕ	{7,2}
7	ϕ	ϕ	{10}
8	ϕ	{9}	ϕ
9	ϕ	ϕ	{10}
10	{11}	ϕ	ϕ
11 *	ϕ	ϕ	ϕ

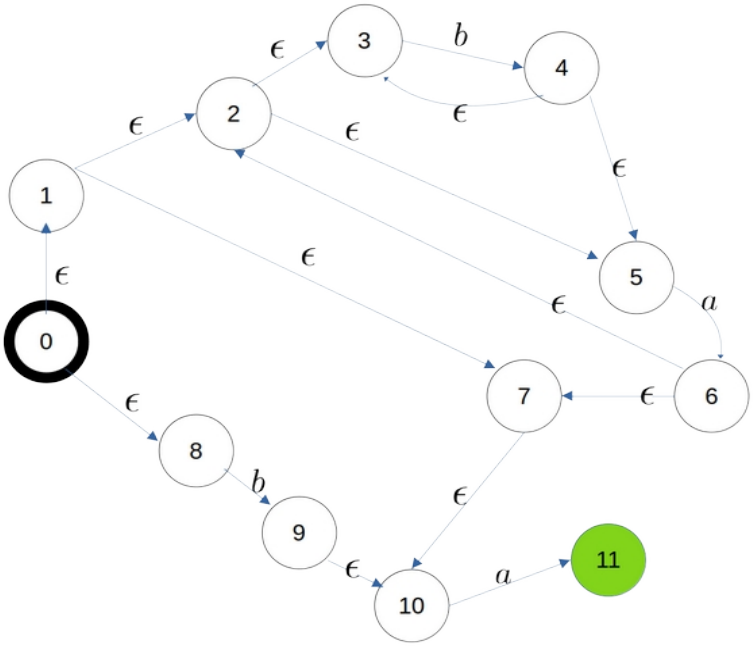

Figura 5: Automata de prueba 1 en forma de grafo dirigido

La salida del programa puede ser alguna de las siguientes:

```
0,1,2,3,4
b,a
0
4
0,b,1
1,b,2
2,b,2
2,a,3
3,b,2
3,a,4
4,b,2
4,a,4
1,a,4
0,a,4
```

Figura 7: AFD equivalente al AFN de prueba 1

```
0,1,2,3,4
a,b
0
1
0,a,1
1,a,1
1,b,2
2,a,3
3,a,1
3,b,2
2,b,2
0,b,4
4,a,1
4,b,2
```

Figura 6: AFD equivalente al AFN de prueba 1

En un principio parecería que no debería de haber dos salidas distintas para un mismo problema, el hecho es que no son distintas, sin embargo este fenómeno ocurrió debido a la implementación del lenguaje *python*, ya que la máquina virtual del lenguaje tiene una semilla aleatoria para funciones *hash*, [3] esto hace que el orden en el que si itera el alfabeto cambie (observe como en ambas salidas el orden en el que se muestra el alfabeto es distinto), y por extensión la numeración de estados cambia, sin embargo, ya se mencionó que estas dos salidas describen el mismo autómata, de hecho se podrán generar 2 grafos isomorfos respecto al otro con estas tablas.

En la siguiente figura se muestra la tabla de transiciones resultante para el nuevo autómata finito y después se muestra el grafo dirigido que representa el nuevo AFD.

Tabla 2: Tabla de transiciones del AFD equivalente al AFN 1

Q	a	b
0	4	1
1	4	2
2	3	2
3	4	2
4	4	2

La siguiente figura muestra el AFD equivalente al AFN en forma de grafo dirigido.

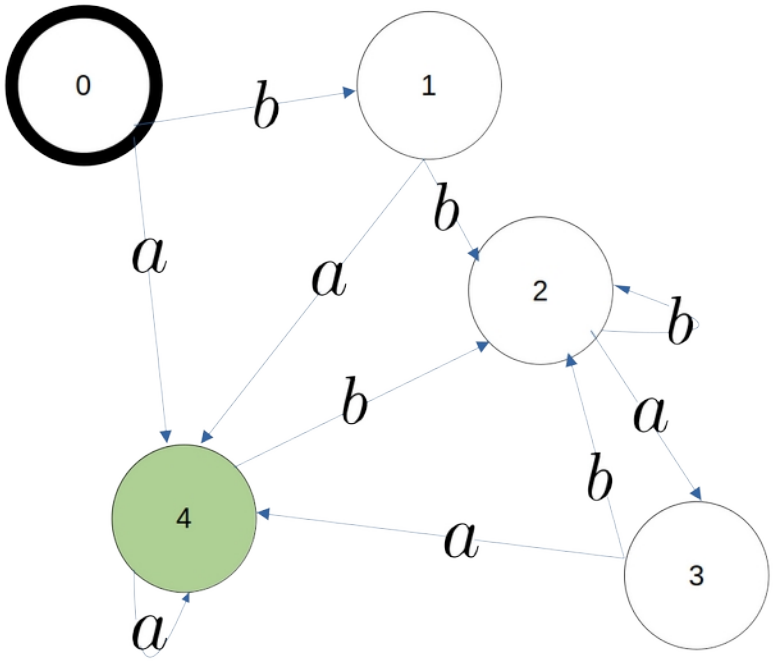

Figura 8: AFD equivalente al AFN1 en su forma de grafo dirigido

La Figura 10 muestra el resultado del programa con el segundo autómata de prueba, la definición del AFN se puede apreciar en Figura 9 y en la Tabla 3

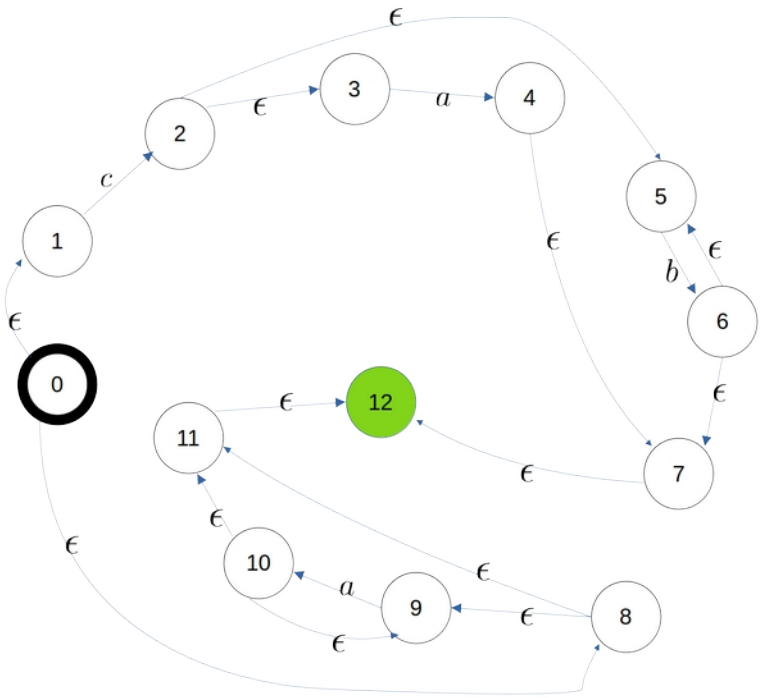

Figura 9: Automata de prueba 2 en forma de grafo dirigido

Tabla 3: Tabla de transiciones del autómata de prueba 2

Q	a	b	c	ϵ
→ 0	ϕ	ϕ	ϕ	{1,8}
1	ϕ	ϕ	{2}	ϕ
2	ϕ	ϕ	ϕ	{3,5}
3	{4}	ϕ	ϕ	ϕ
4	ϕ	ϕ	ϕ	{7}
5	ϕ	{6}	ϕ	ϕ
6	ϕ	ϕ	ϕ	{5,7}
7	ϕ	ϕ	ϕ	{12}
8	ϕ	ϕ	ϕ	{9,11}
9	{10}	ϕ	ϕ	ϕ
10	ϕ	ϕ	ϕ	{9,11}
11	ϕ	ϕ	ϕ	{12}
12*	ϕ	ϕ	ϕ	ϕ

```
0,1,2,3,4,5
b,a,c
0
0,2,4,5
0,b,1
1,b,1
1,a,1
1,c,1
0,a,2
2,b,1
2,a,2
2,c,1
0,c,3
3,b,4
4,b,4
4,a,1
4,c,1
3,a,5
5,b,1
5,a,1
5,c,1
3,c,1
```

Figura 10: AFD equivalente al AFN de prueba 2

Como último punto se mostrará la tabla de transiciones que se puede generar a partir del resultado del programa mostrado en Figura 10.

Q	a	b	c
0	2	1	3
1	1	1	1
2	2	1	1
3	5	4	1
4	1	4	1
5	1	1	1

CONCLUSIÓN

El hecho de poder convertir AFN's a AFD's abre bastantes posibilidades, en especial para describir los autómatas de las expresiones regulares de una manera más concisa en comparación con los autómatas que se generan mediante la construcción de Thompson, otra posibilidad interesante será darle como entrada al generador de AFN's de la práctica pasada la salida de este programa ya que en sí los autómatas finitos no deterministas generalizan a los autómatas finitos deterministas. Tengo que admitir que no conocía muy bien el hecho de que los AFN's tuvieran equivalencias en AFD's, y aunque tengo una idea vaga de exactamente que usos tenga esto, al parecer la bibliografía indica que la parte del análisis léxico de los compiladores suelen tener un enfoque mediante autómatas [2].
Respecto a la implementación de esta práctica debo agregar que si bien conocía el hecho de que *python* tuviese una semilla aleatoria en cada ejecución de su máquina virtual, nunca visualicé que esto podría afectar los resultados de salida de los programas, de hecho cuando me percaté de que habían dos resultados que *parecían* distintos creí que la implementación era incorrecta, sin embargo tras rectificar tal dato, me dí cuenta que la implementación de hecho era correcta, e interesantemente apareció un concepto que creí no ver en práctica, esto fue el isomorfismo de los grafos. Parte de las dificultades que encontré en esta práctica fue reconocer si el programa era correcto, o si mi algoritmo lo era, nuevamente la bibliografía fue de ayuda ya que en una parte de [2] se muestra un algoritmo iterativo para resolver este problema, y la condición para que se detenga es la misma para que la versión recursiva lo haga, esto es, que se repita un subconjunto generado con un símbolo.

REFERENCIAS

[1] J. Hopcroft, Introduction to Automata Theory, Languages, and Computation, 3ra de. Ithaca New York: Prentice Hall, 2006, pp 60-61.

[2] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers Principles, Techniques & Tools, 2da ed. Boston, Massachusetts, USA: Addison-Wesley 2007. pp 152-154.

[3] 1. Command line and environment - Python 3.10.0 documentation, Python Software Foundation, 11 Oct, 2021. Accedido: 11 Oct, 2021. [Online]. Disponible en: https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHASHSEED.