INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

GARZÓN DOMÍNGUEZ GERARDO ISMAEL

2CV13

TEORÍA COMPUTACIONAL

PRÁCTICA 4:
AUTÓMATA FINITO NO DETERMINISTA

INTRODUCCIÓN

Si examinamos los autómatas finitos deterministas, notamos que la característica en común que tienen, es que, la función de transición, δ, define una sola transición para cada par estado-símbolo. En tal caso, la función δ es llamada una función total, y es por ello que estos autómatas son deterministas. [1]

Para los autómatas finitos no deterministas, se agrega la característica de que, para cada estado y símbolo, puede haber más de una transición definida. Formalmente, esto se puede lograr definiendo la función de transición tal que para un par estado-símbolo, esta tenga un rango el cual es un conjunto de estados. [1]

Un autómata finito no determinista esta definido mediante la 5-tupla: $M = (Q, \Sigma, \delta, q_0, F)$. [1]

Dónde:
Q : El conjunto finito de estados internos del autómata.
Σ : El alfabeto de entrada del autómata, toda cadena de entrada debe estar construida sobre este. Para que pueda ser procesada.
δ : La función de transición: $\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$. Una función que mapea un par (q_i, a), $q_i \in Q, a \in \Sigma \cup \{\lambda\}$ a un subconjunto del conjunto potencia de Q.
$q_0 \in Q$: Un elemento perteneciente al conjunto de estados del autómata, se considera como el estado inicial del autómata, es decir desde este estado comenzaría su funcionamiento al procesar una cadena.
$F \subseteq Q$: Subconjunto del conjunto de estados internos del autómata. Este subconjunto representa los estados finales válidos del autómata, es decir que las cadenas que lleven al autómata a este estado como paso final, serán aceptadas. [1]

Como se puede observar, los autómata finitos deterministas, se diferencian de los autómatas finitos no deterministas, por la manera en que está definida la función de transición. A continuación se describirá cual es el mecanismo de operación de un autómata finito no determinista (abreviado como AFN):

De manera similar al AFD, un AFN, se encontrará inicialmente en el estado q_0, a partir de este estado, leerá un símbolo de la cadena y se tomará el resultado de la función δ. Dado que el resultado es un subconjunto de 2^Q, se tomará cada estado del subconjunto como una opción para moverse al siguiente estado, para el cual se repite el proceso (recursividad). Nótese también, que dada la definición de la función δ, se puede aceptar la cadena vacía como argumento para δ. De esta manera el autómata podrá moverse de un estado a otro sin la necesidad de consumir un símbolo de la cadena. [1]

Para representar un AFN mediante un grafo de transiciones $G = (V, E)$, tal que V es el conjunto de vértices del grafo, y E es el conjunto de aristas que unen a los vértices, las aristas pueden ser representadas como pares no ordenados.

Para este caso en particular, si $M = (Q, \Sigma, \delta, q_0, F)$, un autómata finito no determinista. Entonces su grafo de transición asociado G_M será un grafo dirigido que tendrá al menos $|Q|$ vértices, esto es, que los

vértices están definidos por Q, mientras que, las aristas definidas como (q_i, q_j) y etiquetadas con un símbolo $a \in \Sigma \cup \{\lambda\}$, se encontrarán en el grafo si y solo si $q_j \in \delta(q_i, a)$. [1]. Esto implica que en un AFN pueden existir vértices etiquetados como λ, que se les conoce como transición-λ. [1]
El resto de la notación es idéntica a la notación para grafos de AFD's.

Nótese que un diagrama de transiciones se puede generar a partir de la tabulación de la función δ, como ya se menciono dado un par (q_i, a) y $q_j \in 2^Q$. Así que dicha función δ se puede visualizar mediante:

Estados / Símbolo	a
q_i	q_j

Figura 1.0 Forma tabular de una función de transición δ

La tabla se puede expandir para $|Q|$ renglones y $|\Sigma \cup \{\lambda\}|$ columnas.

Por lo tanto una cadena es aceptada por una AFN si hay alguna secuencia de posibles movimientos que lleve al autómata a un estado de aceptación $q_f \in F$ al consumir todos los símbolos de la cadena. Una cadena se rechaza sólo si no hay una secuencia posible de Movimientos mediante tales que lleven al autómata a un estado final valido $q_f \in F$. [1]

La siguiente figura es un ejemplo de un autómata finito no determinista en la forma de un grafo dirigido.

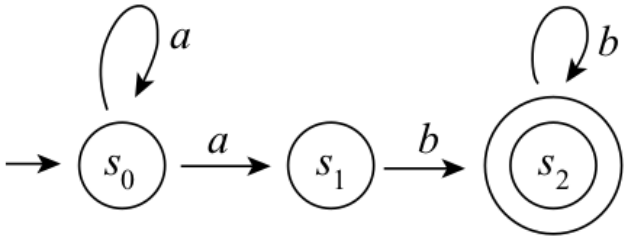

Figura 1.1: Autómata finito no determinista en la forma de un grafo dirigido. [2]

DESARROLLO E IMPLEMENTACIÓN

Para esta práctica se implementará un autómata finito no determinista.
Antes de llegar a tal punto, se explicará cual es el razonamiento detrás de la implementación del tipo abstracto de dato que representa autómatas, y el método que puede operar sobre estas estructuras para recorrer todos los caminos posibles del autómata dada una cadena.

Algunas de las modificaciones realizadas son:

1. Ahora los conjuntos de estados están representados mediante un arreglo dinámico que mantiene registro del número de elementos que contiene. Esto permite iterar sobre los elementos del "conjunto". En este caso podría decir que es más conveniente usar un arreglo como contenedor a otro tipo de estructura, ya que con este se puede indexar en tiempo constante, y en general, número de elementos no crecerá tanto como en otras situaciones (conjuntos de cadenas), de hecho en esta práctica la cantidad de estados es constante siempre. Por tales razones considero que sigue siendo factible tomar tiempo lineal para buscar elementos.
2. La función delta ahora debe regresar forzosamente un conjunto (arreglo) de estados, en cierta manera sigue manteniendo compatibilidad con los AFD's, ya que para un AFD, se considerará que el conjunto (arreglo) regresado, tiene cardinalidad (tamaño) de 1 elemento.
3. Se agregó la función traverse_all_paths, que, como su nombre indica, recorre todos los caminos posibles de un autómata, dada una cadena.
4. El autómata ahora recibe también un apuntador a función, tal que este llama a la función a la que apunta en cada paso que toma el autómata al ejecutar la funcion traverse_all_paths. Con esto se tiene la ventaja de que ahora el autómata se puede comunicar de cierta manera con objetos externos a si, y a partir de esto es que se le puede dar un poco de funcionalidad, (como mantener registro de los caminos que toma) la cual será independiente de como está definido y como funciona una instancia cualquiera del autómata.

A continuación se muestra la nueva definición de la estructura que describe un autómata en el lenguaje C.

```
struct automata_t
{
    strset_t* alphabet;
    ty
    *states,
    *(*Delta)(void* dom_range, const char*),
    *initial_state,
    *final_state;
    void (*step)(str_t*, size_t, void*);
};
```

Figura 2.0: Nueva definición del tipo de dato autómata_t.

La nueva función que se introdujo; *traverse_all_paths*, toma como argumentos un autómata, y una cadena. Esta función hace uso de una función auxiliar para poder resolver el problema de manera recursiva. En este caso elegí la manera recursiva por la sencillez que presenta.

No ocupa espacio extra alojado explícitamente en la función, es decir, el espacio extra que ocupan las variables locales por cada llamada recursiva se aloja automáticamente según el *call stack* (o simplemente *stack*) que genere el compilador [3], así, no se necesita mantener rastro del índice de la cadena de manera explícita, ya que, cuando la función llega al caso base de la recursividad, al retornar, el *stack* va a tener guardados las variables locales anteriores [3] para continuar resolviendo recursiva-mente para cada estado devuelto por la función delta.

Para clarificar lo descrito anteriormente, se mostrará la definición de la función auxiliar recursiva de traverse_all_paths.

```
static void traverse_path
(automata_t* automata, void* curr_state, str_t* expr, size_t curr_idx)
{
    if(curr_idx != str_len(expr))
    {
        array_t* states = automata->Delta(curr_state, str_arr(expr)+curr_idx);
        for(size_t j=0, t=get_size(states); j<t; ++j)
        {
            void* item = get_item(states, j);
            automata->step(expr, curr_idx, item);

            traverse_path(automata, item, expr, curr_idx+1);
        }
    }
}
```

Figura 2.1: Función recursiva para recorrer todos los caminos del autómata, una de sus ventajas, es que no es necesario alojar espacio extra de manera explícita. Como comentario, me gustaría mencionar que esta función es muy parecida a una función que se usaría para recorrer un directorio / carpeta recursiva-mente (cada directorio dentro de cada directorio)

Como se puede apreciar, el caso base es cuando el índice de la cadena es igual a la longitud de la cadena que se le pase, mientras esa condición se cumpla, se tomará el resultado de la función delta para el símbolo y estado actuales, después, para cada elemento del conjunto (arreglo) de estados, se repite la función, pero esta vez con el símbolo siguiente dentro de la cadena. De esta manera, cuando la función *traverse_path*, llegue al caso base por medio de algún camino, regresará a una llamada previa de sí misma, dónde las variables locales han sido guardadas en el *stack* [3]. Desde ese punto, la función debe terminar de ejecutar el ciclo *for*, o sea que, debe resolver para los caminos restantes de la misma manera.
Un detalle a notar, es que el ciclo *for* no se ejecutará las veces el tamaño del arreglo o no se ejecuta en lo absoluto si el arreglo es de tamaño 0, por consecuencia, la función no se vuelve a llamar a sí misma, y regresará a su anterior llamada para resolver para los caminos restantes (si es que existen). Parte de esta definición da versatilidad, ya que el usuario puede definir que la función delta lleve al conjunto que contiene un estado especial, esto para dos casos excepcionales; cuando se tiene un par (estado, símbolo) no definidos por la función de transición, o cuando un par (estado, símbolo) llevan a un conjunto vacío. Definir un conjunto con un estado especial es totalmente opcional, ya que, la función *traverse_path*, sigue siendo válida para conjuntos vacíos.

A continuación, se describirá la implementación del autómata visto en clase.
La función de transición del autómata está dada por la siguiente tabulación:

Estados/símbolos	a	b
$\rightarrow q_0$	$\{q_1, q_4\}$	$\{q_3\}$
q_1	$\{q_1\}$	$\{q_2\}$
$*q_2$	ϕ	ϕ
$*q_3$	ϕ	ϕ
$*q_4$	ϕ	$\{q_4\}$

Figura 2.2: Tabulación de la función de transición del autómata 1 visto en clase.

A continuación se muestran dos casos para cada autómata, uno donde la cadena se acepta, pues se llega a un estado de aceptación, y otro donde no ocurre esto.

```
array_t* delta(void* curr_state, const char* as_arr)
{
    const char opt = *as_arr;
    enum {erri = 5, errf = 8};

    if(opt != 'a' && opt != 'b')
        return set(5);

    if(curr_state == dom(0))
        switch(opt) {
            case 'a': return set(0); case 'b': return set(1);
        }
    else if(curr_state == dom(1))
        switch(opt) {
            case 'a': return set(2); case 'b': return set(3);
        }
    else if(curr_state == dom(2))
        switch(opt) {
            case 'a': return set(erri); case 'b': return set(erri);
        }
    else if(curr_state == dom(3))
        switch(opt) {
            case 'a': return set(erri); case 'b': return set(erri);
        }
    else if(curr_state == dom(4))
        switch(opt) {
            case 'a': return set(erri); case 'b': return set(9);
        }
    else if(curr_state == dom(5))
        switch(opt) {
            default: return set(erri);
        }

    return NULL;
}
```

Figura 2.3: Función delta para el autómata visto en clase, definida en el lenguaje C.

Esta función recibe un estado, encuentra a que elemento del conjunto de estados del autómata corresponde, después decide que conjunto de estados regresar según el símbolo de entrada. El conjunto regresado corresponde al conjunto que se encuentra en algún renglón i, y alguna columna j de la tabla 2.2. En código, este elemento corresponde al elemento $ij-1$ de un arreglo regresado al llamar set($ij-1$).
En este caso, se optó por agregar una serie de conjuntos que contienen un estado especial de error, este conjunto se regresa cuando la función set mapea a elementos del arreglo tales que: $4 \leq ij-1 \leq 9$ y cuando el estado actual sea dicho estado de error, de esta manera, la función delta mapeara al conjunto con el estado de error cuando se encuentre un símbolo inválido o se llegue a una transición no definida para un par (estado, símbolo). De esta manera, la función de la figura 2.1, continuará ejecutando pasos para el resto de símbolos en la cadena, solo que, el resto de pasos se encontrarán en el estado especial de error.

A continuación se mostrarán las pruebas realizadas al autómata usando dos cadenas válidas, las cuales pertenecen a un lenguaje regular. También se realizará la prueba con una cadena inválida, y se explicará por que dicha cadena es inválida.

```
>> aab
(*!)  Camino valido "aab":
>>(0)>>(1)>>(1)>>(2)

Camino invalido "aab":
>>(0)>>(4)>>(Error -1)>>(Error -1)
```

Figura 2.4: Todos los caminos posibles que toma el autómata con la cadena “aab”.

Como se puede notar, la cadena “aab” es aceptada por el autómata, ya que lo lleva a un estado de aceptación, sin embargo, dicha cadena también puede generar un camino inválido dentro del autómata, ya que el estado 4 con el símbolo “a”, mapea al conjunto con estado de error.

```
>> abb
Camino invalido "abb":
>>(0)>>(1)>>(2)>>(Error -1)

(*!)  Camino valido "abb":
>>(0)>>(4)>>(4)>>(4)
```

Figura 2.5: Todos los caminos posibles que toma el autómata con la cadena “abb”.

En este caso, la cadena “abb” es aceptada, ya que existe por lo menos un camino que lleva a un estado de aceptación. Sin embargo, uno de los caminos es inválido, este camino sucede cuando el autómata llega al estado 2, ya que, para el estado 2, cualquier símbolo mapeará al conjunto con el estado de error.

Debido a la definición de la función de transición del autómata, este solo aceptará cadenas que contengan ciertas regularidades. Se aceptarán todas las cadenas construidas sobre el alfabeto {a, b}, las cuales comiencen con cualquier cantidad de a’s, y terminen con una b. O aquellas cadenas que comiencen con una a, y les preceda cualquier cantidad de b’s. Esto se debe principalmente a los estados de pozo en $\delta(q_1, a)$ y en $\delta(q_4, b)$. Esto implica que cualquier cadena que no cumpla con estas condiciones será rechazada, como se muestra en el siguiente ejemplo:

```
>> aabb
Camino invalido "aabb":
>>(0)>>(1)>>(1)>>(2)>>(Error -1)

Camino invalido "aabb":
>>(0)>>(4)>>(Error -1)>>(Error -1)>>(Error -1)
```

Figura 2.6: La cadena “aabb” no genera ningún camino válido para el autómata.

Como se puede apreciar, la cadena “aabb”, no cumple con las regularidades definidas por el autómata, pues esta tiene más de una “a” al inicio.

Como se puede notar, los autómatas tienen cierta relación con las expresiones regulares. Esto se puede explicar mediante el teorema de Kleene el cual afirma que: “Un lenguaje es regular, si y solo si este es aceptado por un autómata”. [2]. Aunque esto no es lo que concierne en esta práctica, sirve como preámbulo para conocer algunas de las aplicaciones de los autómatas.

CONCLUSIÓN

En esta ocasión, se extendió sobre el concepto y teoría de los autómatas finitos deterministas, para describir un nuevo tipo de autómata; un autómata finito no determinista, el cual generaliza a los autómatas finitos deterministas. Esto cuando la función $\delta(q_i, a)$ mapea a un conjunto $q_j \subseteq 2^Q$, y que $|q_j| = 1$.

Desde mi perspectiva, ya es reconocible la relación entre los AFN's y las expresiones regulares, ya que al menos para el autómata desarrollado en esta práctica, existe una expresión regular que de alguna manera describe la función de transición, o también se podría decir que se puede obtener una expresión regular dada la función de transición del autómata.

Una de las que me resultó satisfactoria sobre las práctica, fue que, el problema de recorrer todos los caminos del autómata se prestó para ser resuelto de manera recursiva. Debo reconocer que el problema me resultó un poco más sencillo debido a que la solución tiene bastante similitud con una función que recorrería un directorio de manera recursiva.

Considero que desde el punto de vista de la implementación, esta práctica fue de dificultad media, aunque en cierta manera, menos compleja respecto a las estructuras de datos usadas (comparada con la implementación de conjuntos de cadenas de la práctica 2).

REFERENCIAS

[1] P. Linz, An Introduction to Formal Languages and Automata, 6ta ed. Davis, California: Jones & Bartlett, 2017, pp. 51-52.

[2] J. Anderson, Automata Theory with Modern Applications. 1ra ed. New York: Cambridge University Press 2006, pp. 44, 51.

[3] R. Chang, Lectura de clase, tema: "C Function Call Conventions and the Stack", CMSC313, Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Baltimore. Mar., 2002.