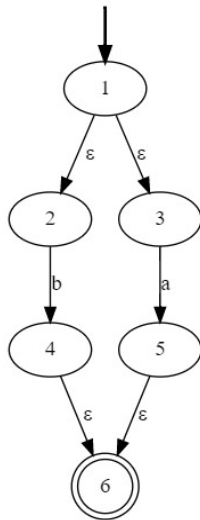


Documentación Expresiones regulares

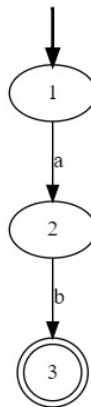
NFA: Algoritmo de Thompson

Para describir los pasos seguidos para resolver la expresión regular por medio de AFN, se siguieron los siguientes pasos:

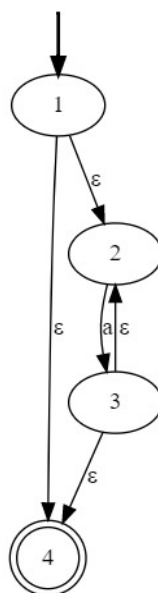
1. El usuario inicialmente ingresa la expresión regular que desea transformar en un autómata finito no determinista.
2. El programa realiza un proceso de "limpieza" que consiste en transformar los operadores $?$ y $+$ en sus respectivas equivalencias ($r? = r|\epsilon$ y $r+ = r * r$).
3. Seguido de esto, el programa realiza un proceso de agregar el operador de concatenación a la expresión regular (ej. $ab \Rightarrow a.b$).
4. Luego, en este punto, la expresión regular cuenta solamente con las 3 operaciones básicas: operación or, operación concat y cerradura kleene. Con esto en mente, se va recorriendo la expresión regular carácter por carácter, diferenciando entre símbolos y operaciones, y dinámicamente, lo transforma en formato postfix a diferencia que las operaciones las realiza en el instante.
5. Las tres operaciones trabajan con 2 posibles operandos, el primero es el símbolo persé y la segunda posibilidad es el resultado de otra operación, una tupla de dos **Nodos**, la característica de esta tupla es que contiene el nodo inicial y el nodo final que se genera al generar el grafo de una operación.
 - a. El resultado de la operación **or** devolvería una tupla con la información del nodo 1 y del nodo 6.



- b. El resultado de la operación **concat** devolvería una tupla con la información del nodo 1 y del nodo 3.



- c. El resultado de la operación **cerradura kleene** devolvería una tupla con la información del nodo 1 y del nodo 4.



6. Previamente a realizar alguna operación, el programa verifica si alguno de los dos operandos (o uno, según sea el caso) es o no una tupla, para realizar las conexiones necesarias entre todos los nodos y finalmente obtener un resultado correcto. La razón de hacer estas verificaciones es porque si alguno de los operandos es una tupla, el programa lo reconoce y sabe que el nodo inicial debe usarlo para ser unido por algún nodo previo, o que el nodo final debe unirse a algún otro nodo, por razones obvias, esas conexiones mencionadas dependen de la operación que se está realizando.
7. Para cada una de las operaciones que se realicen, se generan nodos en dependencia de la operación, y en cada nodo se almacena su nombre (un código único) y las referencias transiciones hacia otros nodos. También, estos nodos generados se almacenan en un listado de estados para su graficación posterior.
8. De este modo, al finalizar todas las operaciones, el único elemento que quedará en la lista de operadores será una tupla con nodo inicial "global", en otras palabras es el estado inicial, y con nodo final, que será el estado de aceptación.

AFD: Algoritmo de subconjuntos

1. Para construir el grafo de un autómata finito determinista, se requiere que el AFN esté ya construido, puesto que trabaja con sus estados y con sus transiciones.
2. Como el algoritmo lo menciona, se deben realizar operaciones de cerradura-epsilon y de la operación **move** de un conjunto de nodos a otros, para esto, se utiliza una clase llamada **DFA_Node** para distinguir un nodo de AFN y un nodo de AFD.
3. Estos nodos de AFD, a diferencia de los nodos AFN, contiene propiedades como un conjunto de identificadores de los nodos de AFN, la propiedad de si está marcado o no (para construir el AFD) y si es un nodo de aceptación.
4. Entonces, para cada cerradura-epsilon que se realice en la construcción de AFD, se genera un DFA_Node con información de todos los nodos AFN que está agrupando.
5. La forma en que se almacenan las transiciones también es diferente en este proceso, ya que en la construcción del AFN se almacena en cada nodo, pero aquí se realiza de forma más simple, una lista con tuplas de 3 elementos:
 - a. Nodo origen
 - b. Símbolo de transición
 - c. Nodo destino

AFD: Algoritmo de construcción directa

1. El proceso empieza como lo hace el algoritmo de AFN, empieza realizando la limpieza que se mencionó anteriormente, y se agregan las operaciones de concatenación donde corresponda.
2. El programa, igualmente, revisa caracter por caracter, diferenciando los símbolos de los operadores.
3. La construcción directa permite dos posibles operadores, un símbolo persé y un objeto de tipo **Leaf**, donde este tipo de dato almacena información como su nombre, su posición dentro del árbol, si es operador o no, los hijos que este contiene, si es anulable o no, su *firstpos* y su *lastpos*.
4. El programa identifica si algún operador (u operadores) son **Leaf** para realizar las operaciones según correspondan para armar el árbol sintáctico. Cada una de las operaciones devuelve una raíz temporal de tipo **Leaf** para las siguientes operaciones, y cuando ya se acaben las operaciones, el único elemento dentro de la lista de operadores será la raíz real del árbol. Cada una de las hojas se almacena en un listado de estados.
5. Desde el momento en que se construye el árbol sintáctico, se van calculando las propiedades de anulable, *firstpos* y *lastpos*, donde, obviamente, depende del operador, símbolo o ϵ .
6. Luego de haber construido el árbol con todas las propiedades mencionadas, se recorre el listado de estados para ser capaces de calcular la propiedad *followpos* del árbol construido, aplicando las reglas para su cálculo correcto.
7. Finalmente, se sigue el algoritmo para construir el AFD a partir del *followpos* calculado, teniendo la misma estructura para las transiciones: una lista con tuplas de 3 elementos:
 - a. Nodo origen
 - b. Símbolo de transición
 - c. Nodo destino

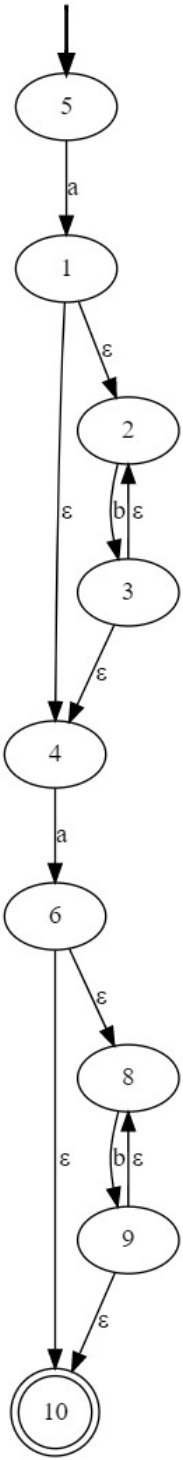
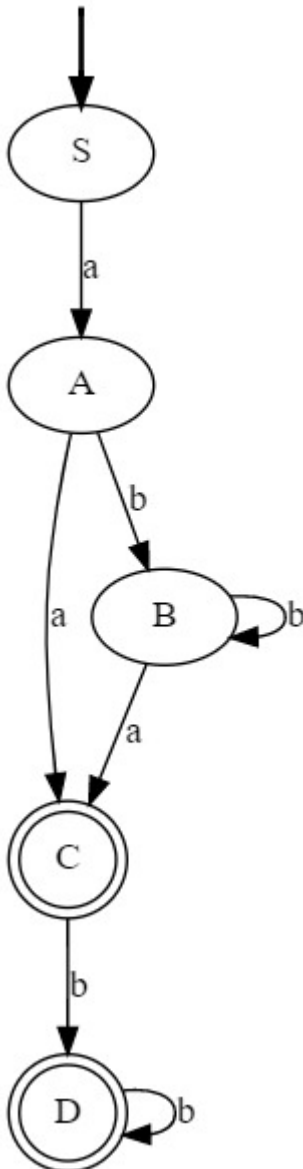
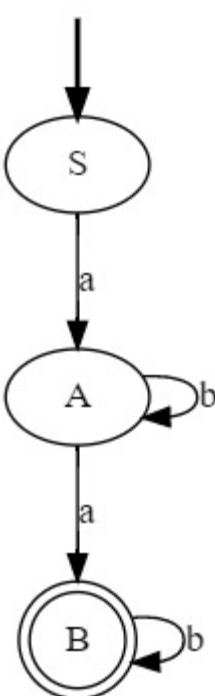
Discusión

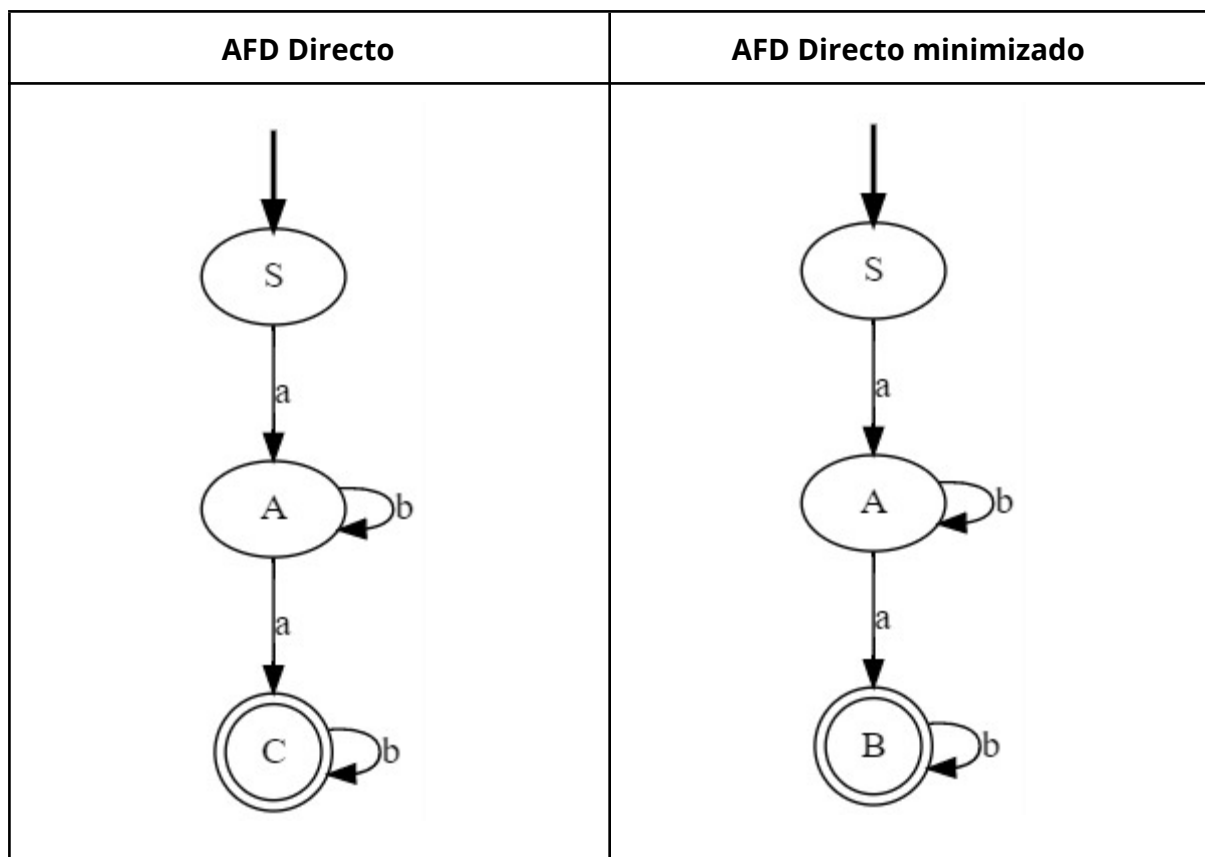
Luego de haber desarrollado las 3 posibles alternativas de para representar las expresiones regulares en grafos, puedo mencionar que el algoritmo más eficiente y más rápido de implementar fue el de construcción de un AFD por método directo. Puesto que los cálculos son fáciles de implementar contra los otros dos, AFN de Thompson y AFD por subconjuntos. Por otro lado, el tiempo de ejecución del método directo es mucho menor a los otros, así que si tuviese que volver a implementar algún algoritmo, sería el de método directo.

Durante la implementación de estos algoritmos, el algoritmo en el que encontré más problemas fue el de AFN de Thompson, puesto que no había tenido tanta relación con grafos y, además, la forma en que se construye es un más manual que el método directo de AFD, porque el programador es el que debe construir el grafo de la operación paso a paso y establecer las transiciones de estas. Porque a diferencia del método directo de AFD, el algoritmo planteaba cálculos de *followpos* para establecer las relaciones entre los nodos y que a veces llegaba al grafo minimizado.

Ejemplos y pruebas

Ejemplo 1 - ab^*ab^*

AFN de Thompson	AFD de subconjuntos	AFD de subconjuntos minimizado
 <p>The Thompson NFA consists of 10 states. State 5 is the start state, indicated by an incoming arrow. State 10 is the final state, indicated by a double circle. Transitions are as follows: 5 to 1 on 'a'; 1 to 2 on ϵ and 1 to 4 on ϵ; 2 to 3 on 'b'; 3 to 2 on ϵ; 3 to 4 on ϵ; 4 to 6 on 'a'; 6 to 8 on ϵ and 6 to 10 on ϵ; 8 to 9 on 'b'; 9 to 8 on ϵ; 9 to 10 on ϵ.</p>	 <p>The subset automaton has five states: S (start), A, B, C (final), and D. Transitions are: S to A on 'a'; A to B on 'b'; B to A on 'a'; A to C on 'a'; B to C on 'a'; C to D on 'b'; D to D on 'b'.</p>	 <p>The minimized subset automaton has three states: S (start), A, and B (final). Transitions are: S to A on 'a'; A to A on 'b' (self-loop); A to B on 'a'; B to B on 'b' (self-loop).</p>



```

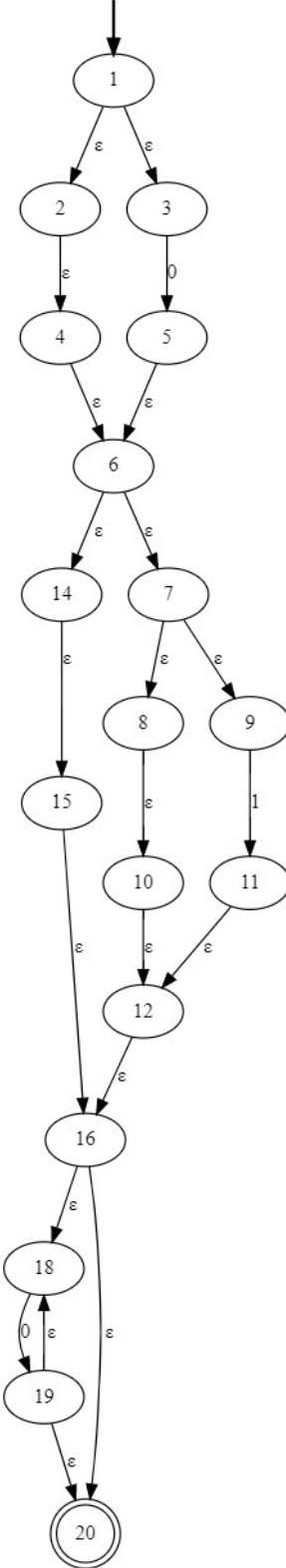
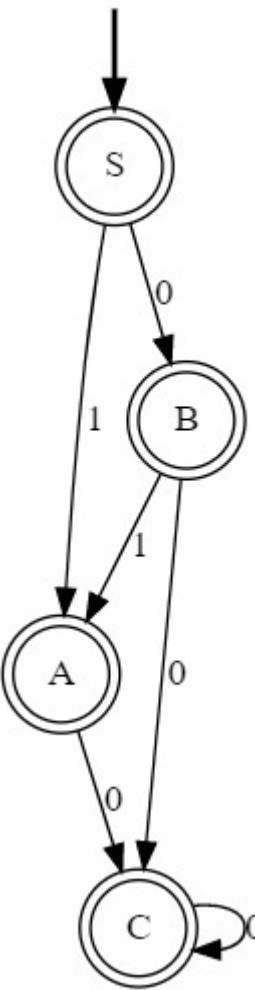
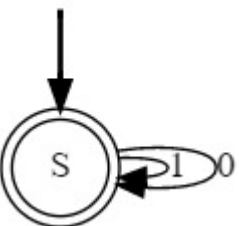
-- AFD DIRECTO --
EXPRESION FIXED: (a.b*.a.b*).#
{1: [1, 3], 2: [1, 3], 3: [4, 5], 4: [4, 5], 5: []}
¿abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb cumple con ab*ab*? yes
-- 0.00002380 seconds --

-- AFN --
EXPRESION FIXED: a.b*.a.b*
¿abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb cumple con ab*ab*? yes
-- 0.00025120 seconds --

-- AFD --
¿abbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb cumple con ab*ab*? yes
-- 0.00004430 seconds --

```

Ejemplo 2 - $0?(1|\epsilon)?0^*$

AFN de Thompson	AFD de subconjuntos	AFD de subconjuntos minimizado
		

AFD Directo	AFD Directo minimizado
<pre> graph TD Start(()) --> S(((S))) S -- 0 --> A(((A))) S -- 1 --> B(((B))) A -- 0 --> B A -- 1 --> B B -- 0 --> B </pre>	<pre> graph TD Start(()) --> S(((S))) S -- "0 1" --> S </pre>

```

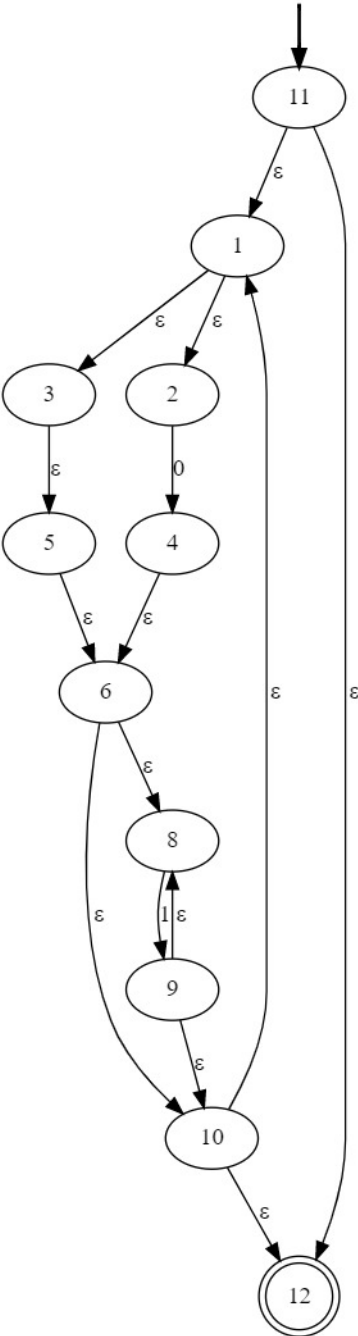
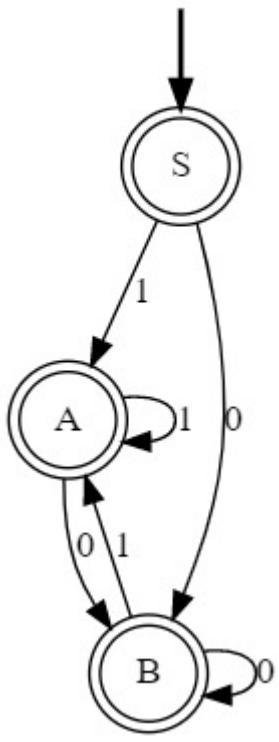
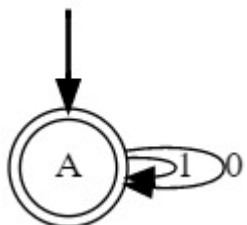
-- AFD DIRECTO --
EXPRESSION FIXED: ((0|ε).((1|ε)|ε).0*).#
{1: [2, 3, 4], 2: [3, 4], 3: [3, 4], 4: []}
¿000000 cumple con 0?(1|ε)?0*? yes
-- 0.00000950 seconds --

-- AFN --
EXPRESSION FIXED: (0|ε).((1|ε)|ε).0*
¿000000 cumple con 0?(1|ε)?0*? yes
-- 0.00018820 seconds --

-- AFD --
¿000000 cumple con 0?(1|ε)?0*? yes
-- 0.00001500 seconds --

```

Ejemplo 3 - $((\epsilon|0)1^*)^*$

AFN de Thompson	AFD de subconjuntos	AFD de subconjuntos minimizado
 <p>The Thompson NFA diagram consists of 12 states. State 11 is the start state, indicated by an incoming arrow. State 12 is the final state, indicated by a double circle. Transitions are as follows: 11 to 1 (labeled ε), 1 to 3 (labeled ε), 1 to 2 (labeled ε), 3 to 5 (labeled ε), 2 to 4 (labeled 0), 5 to 6 (labeled ε), 4 to 6 (labeled ε), 6 to 8 (labeled ε), 8 to 9 (labeled 1), 9 to 8 (labeled ε), 8 to 10 (labeled ε), 9 to 10 (labeled ε), 10 to 12 (labeled ε), 1 to 12 (labeled ε), and 11 to 12 (labeled ε).</p>	 <p>The subset automaton has three states: S (start state), A, and B. S is the start state with an incoming arrow. A and B are final states, indicated by double circles. Transitions are: S to A (labeled 1), A to A (labeled 1), A to B (labeled 0), B to A (labeled 1), B to B (labeled 0), and S to B (labeled 0).</p>	 <p>The minimized subset automaton has a single state A, which is both the start state (indicated by an incoming arrow) and the final state (indicated by a double circle). There is a self-loop on state A labeled with both 1 and 0.</p>

