DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Parte 1

tlengrep

October 13, 2023 Teoría de Lenguajes

Integrante	LU	Correo electrónico
Santiago Fiorino	516/20	fiorinosanti@gmail.com
Lucas Mas Roca	122/20	lmasroca@gmail.com
Juan Pablo Lebon	228/21	juanpablolebon98@gmail.com

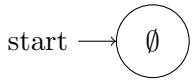
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

1 Construcción de Autómatas Finitos No-Determinísticos

El primer paso en este proceso de obtención de un AFD mínimo para una expresión regular es la obtención de uno no determinístico. Para esto, desarrollamos código para cada operación posible en una expresión regular (las cuales detallamos a continuación), que implementa las técnicas teóricas vistas en clase para pasar de estas operaciones a un autómata que las compute.

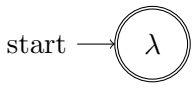
1.1 Expresión regular $\emptyset$

Un autómata no acepta nada si no tiene estados finales. Para construir un autómata para $\emptyset$, entonces, podemos simplemente crear un estado inicial que no sea final.



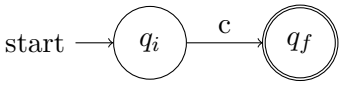
1.2 Expresión regular λ

Para construir un autómata que compute λ, basta con inicializar un nodo que sea inicial y final a la vez, sin ninguna transición. Lo único que puede hacer este autómata es terminar, con lo cual sólo aceptará la cadena λ.



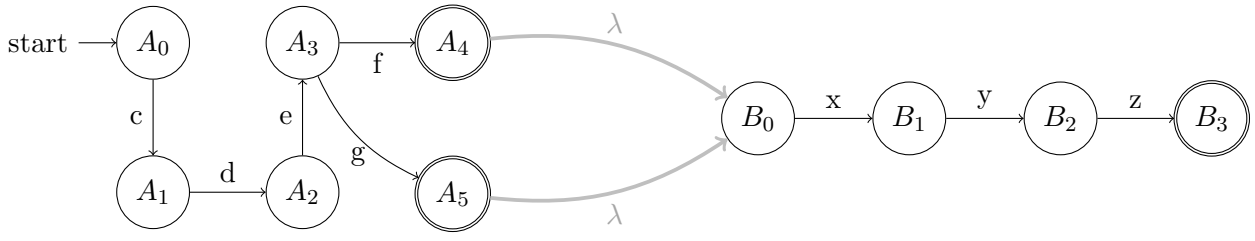
1.3 Expresión regular para c $\in \Sigma$

Un autómata que acepta un caracter c de un alfabeto Σ es simplemente uno con dos estados, uno inicial y uno final, y una transición que va del inicial al final consumiendo c.



1.4 Concatenación de expresiones regulares

Como vimos en la materia, para construir un autómata que compute la concatenación de dos expresiones regulares podemos, a partir de autómatas que corresponden a cada expresión, concatenarlos de la siguiente forma:

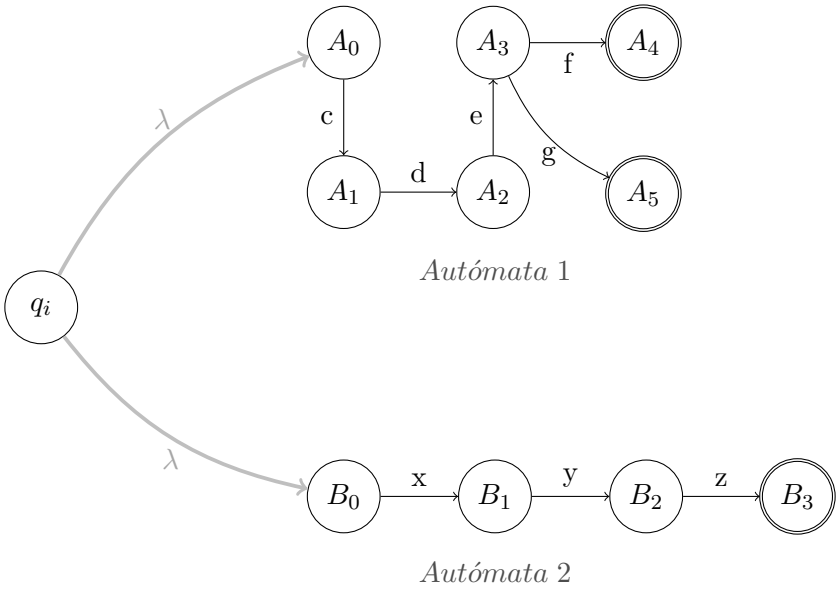


Autómata 1 *Autómata* 2

Es decir, conectamos todos los estados finales del primer autómata con el estado inicial del segundo, mediante transiciones lambda.

1.5 Unión de dos autómatas

También vimos en clase un algoritmo para construir un autómata para la unión de dos expresiones regulares, partiendo de autómatas para cada expresión:



Autómata 1

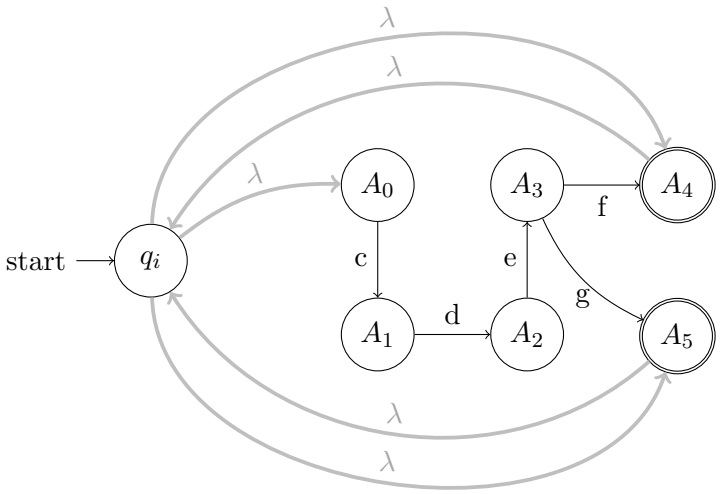
Autómata 2

Es decir, creamos un nuevo estado inicial, y mandamos desde él transiciones lambda a los antiguos estados iniciales de ambos autómatas.

1.6 Clausura de Kleene (Operador $*$)

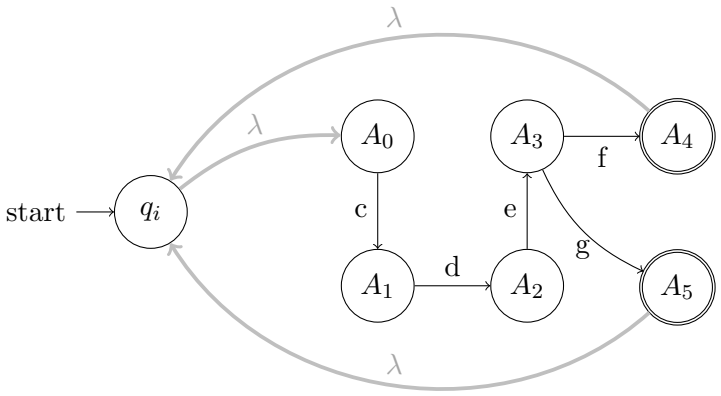
Para computar la clausura de Kleene de una expresión regular R, a partir de un autómata para R, podemos hacer lo siguiente:

1. Crear un nuevo estado inicial q_i. Conectarlo mediante una transicion λ al antiguo estado inicial del autómata, para poder aceptar todo lo que el autómata aceptaba originalmente.
2. Mandar transiciones λ desde los estados finales del autómata original a q_0, para poder recorrer el autómata una cantidad arbitraria de veces.
3. Mandar transiciones λ desde q_i hacia los estados finales. Esto es necesario para poder aceptar la cadena vacía aún si el autómata original no lo hacía.



1.7 Operador $+$

Para este operador podemos hacer algo muy similar al caso anterior, la única diferencia siendo que el estado inicial nuevo q_i no llega directamente a los estados finales mediante transiciones λ. Así, podemos estar seguros que para aceptar una cadena debemos haber recorrido el autómata al menos una vez.



2 Determinización y Minimización

Para pasar del AFND obtenido con los métodos descriptos en la sección anterior a uno determinístico y mínimo, usamos los algoritmos vistos en la materia. Cabe destacar que si bien el algoritmo de minimización requiere un autómata completo y sin estados inalcanzables, en nuestra implementación no chequeamos que se cumpla esto al momento de minimizar. Esto es simplemente porque al primero determinizar el autómata recibido, devolvemos uno que es además completo y cuyos estados son todos alcanzables desde el inicial.

3 Aceptación de caracteres (Función `Match`)

Para implementar una manera de decidir si una cadena α pertenece a un lenguaje regular R, desarrollamos el siguiente algoritmo:

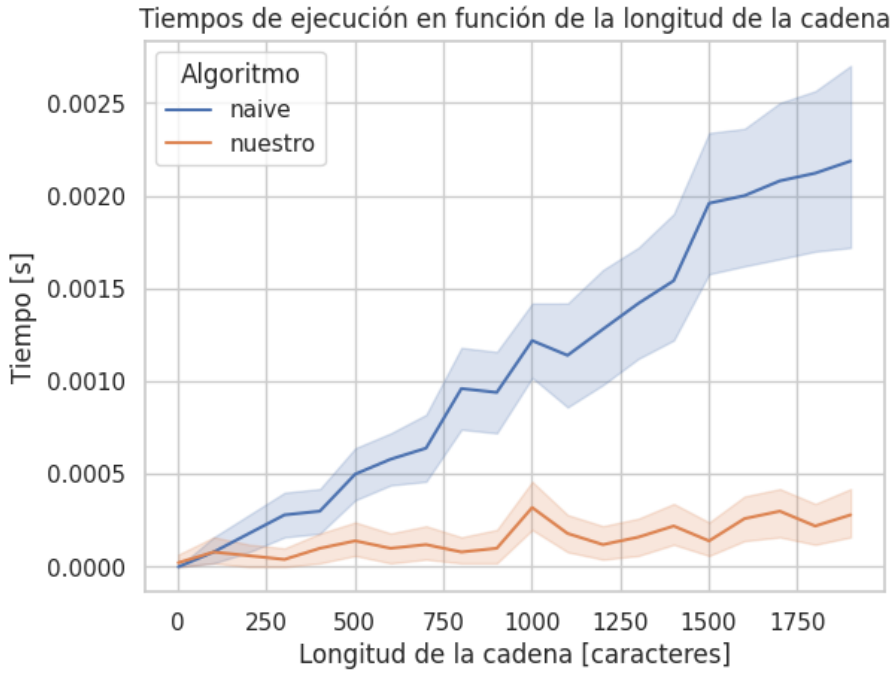
1. Si ya se construyó previamente un AFD mínimo para R, ir al paso 4
2. Armar un AFND M para R
3. Determinizar y luego minimizar M
4. Recorrer el autómata minimizado, partiendo desde el estado inicial y consumiendo cada caracter de α con la transición que permita hacerlo.
5. Si en algún momento llegamos a un estado del cual no podemos salir, porque ninguna transición permite consumir el caracter actual, retornamos False. Si llegamos a un estado final habiendo consumido todo α, retornamos True.

Cabe destacar que, como se espera consultar por la pertenencia de muchas cadenas al lenguaje generado por una misma expresión regular R, guardamos el autómata asociado a R la primera vez que lo construimos, ya que este es sin duda el paso más caro del algoritmo de aceptación. Así, cada futura consulta tendrá costo lineal en la longitud de la cadena, ya que solo se deberá recorrer el autómata consumiendo cada caracter.

4 Experimentación

4.1 Tamaño de la cadena

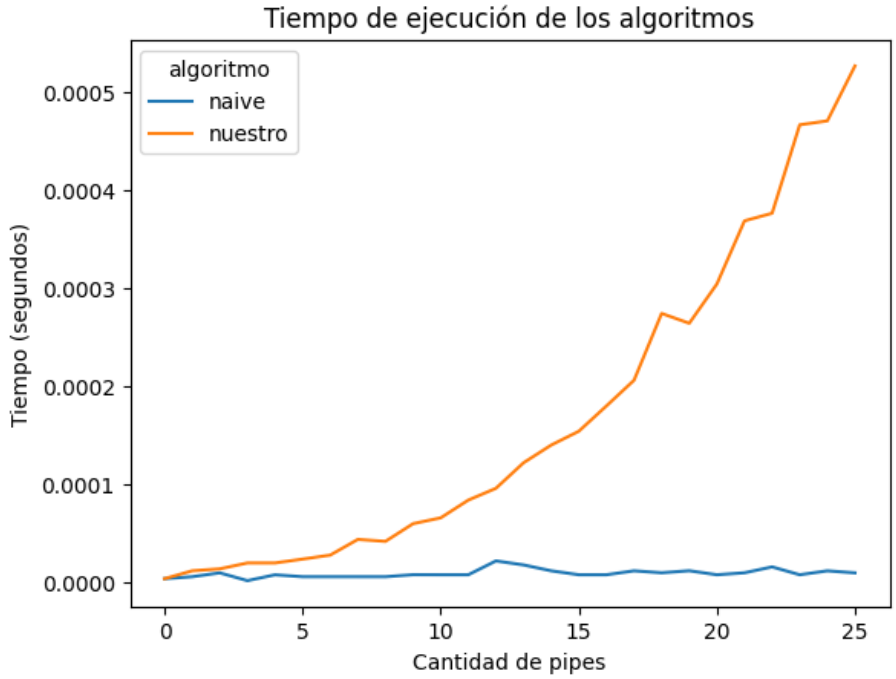
Un parámetro que variamos para testear la performance de ambos algoritmos es el tamaño de las cadenas a matchear:



Podemos ver que al incrementar la longitud de las cadenas, el tiempo de ejecución del algoritmo naive incrementa rápidamente, mientras que el tiempo de ejecución del algoritmo nuestra muestra incrementos muy leves.

4.2 Tamaño del autómata

Finalmente, buscamos testear el tiempo de construcción de un autómata a partir de una expresión regular, dado que es de interés ver cuánto tarda el proceso de determinización y minización. Para esto nuevamente buscamos definir expresiones regulares cada vez más extensas (en este caso, más símbolos |).



Acá se nota el gran costo de construir un autómata con las propiedades buscadas. Sin embargo, considerando la tabla anterior, vemos que esta inversión inicial de tiempo vale la pena a la larga, ya que al tener construido el autómata, matchings subsecuentes son mucho más rápidos con nuestra implementación en general.