Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2223 - Teoría de Autómatas y Lenguajes Formales

Ayudantía 5

Franco Bruña y Dante Pinto
10 de Septiembre, 2021

Pregunta 1

Sea L un lenguaje regular sobre el alfabeto Σ. Demuestre que el siguiente lenguaje:

$$L^{\exists n} = \{w \in \Sigma^* \mid \exists n \in \mathbb{N}.\ w^n \in L\}$$

es regular usando autómatas finitos en dos direcciones.

Observando el lenguaje $L^{\exists n}$ y comparándolo con el lenguaje original L, podemos ver que las palabras w en $L^{\exists n}$ son tales que al concatenarlas una cantidad arbitraria (n) de veces consigo mismas, estas forman una palabra de L.

Además, sabemos que L es un lenguaje regular, lo que significa que existe un DFA A que lo define. Por lo anterior, intuitivamente, podemos pensar que el autómata para $L^{\exists n}$ deberá ejecutar el autómata original A sobre la palabra w y, si llega al final de la palabra, deberá volver a leerla, continuando la ejecución desde donde se quedó (una cantidad arbitraria de veces).

Dado que estamos trabajando con autómatas en dos direcciones, otra manera de ver lo anterior es pensar que ejecutaremos el autómata A hasta que se acabe la palabra, "pausaremos" la ejecución, haremos *rewind* de la palabra hasta volver al principio y continuaremos la ejecución.

Para hacer lo anterior, crearemos una copia de cada estado del autómata, que representarán los estados *rewind*, y haremos que desde cualquier estado, al leer la marca final, el autómata pase al estado *rewind* equivalente, retroceda hasta leer la marca inicial y luego vuelva al estado original para continuar la ejcución.

La construcción asociada a esto, asumiendo que $A = (Q, \Sigma, \delta, q_0, F)$ es:

$A' = (Q', \Sigma, \vdash, \dashv, \delta', q_0, F'$
$Q' = Q \uplus Q_r \uplus q_f \quad \wedge \quad Q_r = q_r \mid q \in Q$
$F' = \{q_f\}$
Sea $a \in \Sigma$, δ' :

- $\delta'(q_0, \vdash) = (q_0, \rightarrow)$
- $\delta'(q, a) = (\delta(q, a), \rightarrow)$
- $\delta'(q, \vdash) = (q_r, \leftarrow),\ q \notin F \quad \wedge \quad \delta'(q, \vdash) = (q_f, \rightarrow),\ q \in F$
- $\delta'(q_r, a) = (q_r, \leftarrow)$
- $\delta'(q_r, \vdash) = (q, \rightarrow)$

Finalmente, podemos demostrar que el lenguaje que define A es equivalente a $L^{\exists n}$ de la forma tradicional (usando las ejecuciones). Para una demostración completa de este proceso, pueden revisar la pregunta 2 de la ayudantía 3.

¿El autómata encontrado termina su ejecución para todas las palabras?. Si no es el caso, diseñe un algoritmo que reciba el $2DFA$ y una palabra w como input y retorne `TRUE` si el autómata acepta la palabra y `FALSE` en caso contrario.

Podemos ver que nuestro autómata no terminará sus ejecuciones para las palabras que no pertenecen al lenguaje.

Para diseñar el algoritmo que buscamos, tenemos que analizar las ejecuciones del $2DFA$. Observándolas, podemos notar que para que el autómata se quede ejecutando por siempre, esto debe ocurrir en un ciclo, pues tenemos una cantidad finita de estados, así que podríamos terminar nuestro algoritmo cuando nos topemos un ciclo, pero antes de decidir si hacer esto, debemos tener cuidad con no eliminar ejecuciones de aceptación.

Considere entonces una ejecución de acpetación cualquiera ρ, que tenga un ciclo. Esto implica entonces que en algún momento de la ejecución llegamos a una configuración (q_c, i_c), luego pasamos por otras configuraciones, volvemos a (q_c, i_c) y de ahí en adelante continuamos la ejecución hasta aceptar la palabra. Dado que q_c representa un estado, i_c representa una posición del lector y estamos trabajando con un autómata determinista; lo anterior no puede ocurrir, pues para cada estado q_c, δ definirá una única posible transición leyendo la letra de la posición i_c, por lo que de encontrar un ciclo en la ejecución, nos quedaremos en el para siempre.

Habiendo encontrado lo anterior, podemos escribir el siguiente algoritmo:

```
Eval(A, w){
q=q_0; i=1; conf={(q, i)}
while True:
    if q in F AND i=n+1:
        return True
    q, d = delta(q, a_i)
    i+=d
    if (q, i) in conf:
        return False
    conf.add(q, i)
}
```

Pregunta 2 (I1 2018)

Considere el siguiente problema:

Problema:	#DFA
Input:	Un DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ y 0^n.
Output:	$\lvert\{w \in \mathcal{L}(\mathcal{A}) \mid \lvert w\rvert = n\}\rvert$

En otras palabras, el problema #DFA consiste en, dado un autómata finito determinista $\mathcal{A}$ y dado una palabra de largo n, contar todas las palabras de largo n que acepta $\mathcal{A}$.

Escriba un algoritmo que resuleva #DFA en tiempo $\mathcal{O}(|\mathcal{A}| \cdot n)$ donde $|\mathcal{A}|$ es el número de estados y transiciones de $\mathcal{A}$. Demuestre la correctitud de su algoritmo.

Una posible solución consiste en modificar el algoritmo de evaluación de NFA *on-the-fly* para que guarde para cada estado la cantidad de ejecuciones que han llegado hasta el en i pasos. EL siguiente algoritmo ejemplifica la idea:

Function `DFANumOfWords`$(\mathcal{A}, n)$

$S := \varnothing$; $S_{\text{old}} := \varnothing$
$S(q_0) := 1$
foreach $q \in Q \setminus \{q_0\}$ **do**
  $S(q) := 0$
for $i := 1$ **to** n **do**
  $S_{old} := S$
  foreach $q \in Q$ **do**
    $S(q) := 0$
  foreach $(p, q) \in \{(p, q) \mid p, q \in Q.\ \exists a \in \Sigma.\ \delta(p, a) = q\}$ **do**
    $S(q) := S(q) + S_{old}(p)$
return $\sum_{q \in F} S(q)$

En efecto, el algoritmo posee complejidad $\mathcal{O}(|\mathcal{A}| \cdot n)$ ya que en el peor caso hay que recorrer todas las transiciones y estados de $\mathcal{A}$ por cada uno de los n pasos.
Ahora, una forma de demostrar la correctitud del algoritmo es la siguiente. Definamos:

$$S_{qi} = |\{w = a_1 a_2 \dots a_n \in \Sigma^* \ \mid \exists \rho_{\mathcal{A}} : q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \dots \xrightarrow{a_i} q\}|$$

Es decir, el tamaño del conjunto de todas las palabras con ejecuciones parciales de tamaño i sobre $\mathcal{A}$ que llegan al estado q_i. Luego queremos demostrar que, en la iteración i-ésima del algoritmo se cumple lo siguiente:

$$S(q) = S_{qi},\ \forall q \in Q$$

Una forma de hacer esto es hacer inducción sobre los pasos del **for** del algoritmo. El primer caso base sería cuando $i = 1$, por lo que queremos demostrar lo siguiente:

$$S(q) = S_{q1},\ \forall q \in Q$$

Podemos notar que en este paso $S_{old} = \{q : 0 \mid \forall q \in Q \setminus \{q_0\}\} \cup \{q_0 : 1\}$ y se cumple que:

$$S(q) = \sum_{\substack{(p,q) \\ \exists a \in \Sigma.\ \delta(p,a)=q \\ p=q_0}} S_{old}(p)$$

Es decir, por cada transición desde q_0 a algún estado agregamos 1 a la suma, por lo que efectivamente contamos las palabras de tamaño 1.
Y por ende se cumple que:

$$S_{q1} = |\{w = a_1 a_2 \dots a_n \in \Sigma^* \ \mid \exists \rho_{\mathcal{A}} : q_0 \xrightarrow{a_1} q\}|$$

Otro caso base es cuan tenemos estados de origen aparte del inicial ($i = 2$). De manera casi análoga al caso anterior podemos notar que:

$$S(q) = \sum_{\substack{(p,q) \\ \exists a \in \Sigma.\ \delta(p,a)=q}} S_{old}(p)$$

Es decir, por cada estado p desde el que podemos llegar a q con una letra vamos a sumar $S_{old}(p)$ lo que corresponde a todas las palabras de largo 1 que se formaban llegando a p. Esto ya que por cada una de esas formamos una nueva con esta letra de la transición.
Ahora para el caso inductivo asumimos que S_{qi} cumple con lo que buscamos para todo q. Ahora, podemos ver que para la iteración $i + 1$

$$S(q) = \sum_{\substack{(p,q) \\ \exists a \in \Sigma.\ \delta(p,a)=q}} S_{pi}$$

Lo que equivale a:

$$S(q) = \sum_{\substack{(p,q) \\ \exists a \in \Sigma.\ \delta(p,a)=q}} |\{w = a_1 a_2 \dots a_n \in \Sigma^* \ \mid \exists \rho_{\mathcal{A}} : q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \dots \xrightarrow{a_i} p\}|$$

Por nuestra hipótesis de inducción.
Finalmente como el algoritmo devuelve la siguiente expresión:

$$\sum_{q \in F} S_{qn}$$

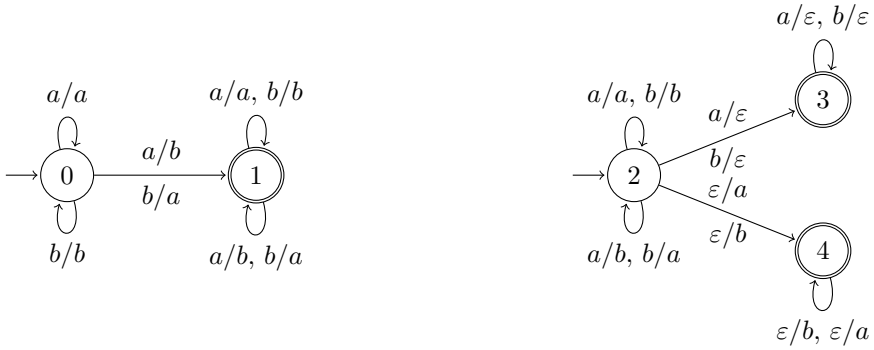
podemos notar que retorna la cantidad de palabras que se formaron desde q_0 hasta cada estado final en n pasos.

Pregunta 3

Sea $\Sigma = \{a, b\}$. Para cada una de las siguientes relaciones construya un transductor equivalente:

1. El complemento de la identidad entre palabras de Σ^*, representado por $\neq$.

   Podemos dividir el que dos palabras no sean idénticas en 2 casos, las palabras tienen distinto largo o tienen el mismo largo y al menos una letra diferente. Siguiendo esta idea, podemos crear un transductor no determinista creando ramas disjuntas para cada una de estas condiciones, obteniendo:



   Cabe señalar que al construir transductores (y autómatas) no deterministas con componentes disjuntas, hay que tener cuidado con que las distintas ramas no agreguen palabras que queremos evitar.

   En este caso no hay problema, pues la primera rama del transductor siempre lee y escribe a la vez, por lo que relacionará palabras del mismo largo y llegará a un estado final solamente si al menos una de sus letras es diferente; mientras que la segunda componente dejará eventualmente de leer **o** de escribir y se quedará en un estado final que solo le permitirá continuar escribiendo o leyendo (respectivamente), garantizando que el largo de ambas palabras es diferente.

2. La relación de "orden lexicográfico" $\preccurlyeq$ entre palabras de Σ^*, es decir

$$u \preccurlyeq v \iff \begin{cases} v = uw & \text{para } w \in \Sigma^* \\ u = x \cdot a \cdot y \ \wedge \ v = x \cdot b \cdot z & \text{para } x, y, z \in \Sigma^* \end{cases}$$

   Tenemos dos casos para el orden lexicográfico, que la palabra u sea prefijo de la palabra v o que ambas palabras sean iguales hasta que en cierto punto u tenga una a y v tenga una b (Este orden funciona de manera muy similar al orden alfabético).

   Podemos ver que el segundo caso es muy parecido a nuestro primer caso de la sección anterior, solo que en vez de buscar que ambas palabras tengan una letra diferente, buscamos específicamente que u tenga una a y v tenga una b en esa misma posición y no nos importa el largo luego de esto. Similarmente, la primera condición es muy similar a nuestro segundo caso del paso anterior, solo que esta vez buscamos que u sea más corta que v y además que sus primeras letras sean iguales. Modificando el transductor anterior de acuerdo a los cambios mencionados, encontramos:

3. La relación de "orden *radix*" $\sqsubseteq$ entre palabras de Σ^*, es decir

$$u \sqsubseteq v \iff \begin{cases} |u| < |v| \\ |u| = |v| \ \wedge \ u \preccurlyeq v \end{cases}$$

Nuevamente esta relación es muy similar a las anteriores. La primera condición exige que u sea más corta que v (Una mezcla de las condiciones 2 y 1 anteriores, respectivamente) y la segunda nos dice que si tienen el mismo largo, u debe ser menor a v según orden lexicográfico, lo que significa que deben ser iguales hasta un punto dónde u debe tener una a y v una b (La mezcla de las condiciones 1 y 2 anteriores, respectivamente). Modificando los transductores anteriores, tenemos: