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

IIC2223 – Teoría de Autómatas y Lenguajes Formales – 2' 2024
IIC2224 – Autómatas y Compiladores

Tarea 3

Publicación: Viernes 4 de octubre.
Entrega: **Jueves 10 de octubre hasta las 23:59 horas**.

Indicaciones

- Debe entregar una solución para cada pregunta (sin importar si está en blanco).
- Cada solución debe estar escrita en LaTeX. No se aceptarán tareas escritas a mano ni en otro sistema de composición de texto.
- Responda cada pregunta en una hoja separada y ponga su nombre en cada hoja de respuesta.
- Debe entregar una copia digital por el buzón del curso, antes de la fecha/hora de entrega.
- **Se penalizará con 1 punto en la nota final de la tarea por cada regla que no se cumpla.**
- La tarea es individual.

Pregunta 1

Usando el teorema de Myhill-Nerode, demuestre que el siguiente lenguaje no es regular:

$$\text{Mod} \;=\; \{a^i b^j c^{i \bmod j} \in \Sigma^+ \mid i \geq 0 \wedge j \geq 2\}.$$

Solución

El teorema de Myhill-Nerode nos dice que un lenguaje L es regular ssi existe un número finito de clases de equivalencia bajo la relación de equivalencia $\equiv_L$. Dadas $u, v \in \Sigma^*$, decimos que:

$$u \equiv_L v \iff (\forall w \in \Sigma^*.\;\; u \cdot w \in L \iff v \cdot w \in L)$$

Para demostrar que Mod no es regular, podemos utilizar el contra positivo del teorema de Myhill-Nerode. Es decir, si encontramos una cantidad infinita de clases de equivalencia bajo $\equiv_{\text{Mod}}$, entonces Mod no es regular. Para que dos palabras $u, v \in \Sigma^*$ no sean equivalentes bajo $\equiv_{\text{Mod}}$, basta notar que:

$$(u \not\equiv_L v) \iff (\exists w \in \Sigma^*.\;\; u \cdot w \in \text{Mod} \;\wedge\; v \cdot w \notin \text{Mod}) \text{ (o viceversa).}$$

Dado $n \geq 1$, definimos $u_n = a^n b^{2n}$ y planteamos su clase de equivalencia $C_n = [u_n]_{\equiv_L}$.

Por demostrar. $\forall n < m.\; [u_n]_{\equiv_{\text{Mod}}} \neq [u_m]_{\equiv_{\text{Mod}}}$.

Para esto, basta demostrar que $u_n \not\equiv_{\text{Mod}} u_m$ para todo $n \neq m$. Como $n < m$, escogemos $w = c^n$ y se tiene que:

- $u_n \cdot w = a^n b^{2n} c^n = a^n b^{2n} c^{n \bmod 2n} \in \text{Mod}$.
- $u_m \cdot w = a^m b^{2m} c^n = a^m b^{2m} c^{n \bmod 2m}$. Pero $m \neq n \bmod 2m$, ya que $n < m$, por lo tanto, $u_m \cdot w \notin \text{Mod}$.

Por lo tanto, cada u_n define una clase de equivalencia distinta, y dado que hay una cantidad infinita de naturales, el lenguaje Mod no es regular.

Distribución de puntaje

- 1 punto por plantear cómo usar el teorema de Myhill-Nerode.
- 1 punto por definir u_n y C_n.
- 1 punto por la elección de un w adecuado.
- 1 punto por demostrar que existen infinitas clases de equivalencia.

Pregunta 2

Sea $\mathcal{A} = (Q, \Sigma, \vdash, \dashv, \delta, q_0, F)$ un autómata finito determinista en 2 direcciones (2DFA). Para una palabra $u \in \Sigma^*$ cualquiera, recuerde que $T_u : Q \cup \{\circ\} \to Q \cup \{\perp\}$ es una función tal que:

- para todo $p \in Q$, $T_u(p) = q$ si desde la configuración $(p, |u|)$ el 2DFA $\mathcal{A}$ cruza a la configuración $(q, |u|+1)$ por primera vez, esto es, para algún $m \geq 0$ se tiene que $(p, |u|) \xrightarrow{\mathcal{A}} (q_1, i_1) \xrightarrow{\mathcal{A}} \ldots \xrightarrow{\mathcal{A}} (q_m, i_m) \xrightarrow{\mathcal{A}} (q, |u|+1)$ con $q_k \in Q$ y $i_k \leq |u|$ para todo $k \leq m$. En otro caso, se define $T_u(p) = \perp$ (nunca cruza hacia la posición $|u|+1$).
- $T_u(\circ) = q$ si desde la configuración inicial $(q_0, 0)$ el 2DFA $\mathcal{A}$ cruza a la configuración $(q, |u|+1)$ por primera vez, esto es, para algún $m \geq 0$ se tiene que $(q_0, 0) \xrightarrow{\mathcal{A}} (q_1, i_1) \xrightarrow{\mathcal{A}} \ldots \xrightarrow{\mathcal{A}} (q_m, i_m) \xrightarrow{\mathcal{A}} (q, |u|+1)$ con $q_k \in Q$ y $i_k \leq |u|$ para todo $k \leq m$. En otro caso, se define $T_u(\circ) = \perp$ (nunca cruza hacia la posición $|u|+1$).

Para cada una de las siguientes preguntas escriba un algoritmo y argumente su correctitud.

1. Escriba un algoritmo en pseudocódigo que reciba como entrada un 2DFA $\mathcal{A}$, una función $T_u : Q \cup \{\circ\} \to Q \cup \{\perp\}$ y una letra $a \in \Sigma$ y entregue como output la función T_{ua}. Su algoritmo debe tomar tiempo a lo más $O(|\mathcal{A}|)$.
2. Escriba un algoritmo en pseudocódigo de evaluación de 2DFA sobre palabras haciendo *una sola pasada* sobre la palabra de entrada. Específicamente, su algoritmo recibe como entrada un 2DFA $\mathcal{A}$ y una palabra w, y debe retornar `true` sí, y solo sí, $w \in \mathcal{L}(\mathcal{A})$. Su algoritmo deberá tomar tiempo $O(|\mathcal{A}| \cdot |w|)$ y deberá hacer una sola pasada sobre w, esto es, puede procesar la palabra w de izquierda a derecha una y sola una vez. En particular, su algoritmo esta restringuido en que NO puede iterar sobre la palabra multiples veces, ni hacer una copia de ella.

   Para esta pregunta usted puede asumir que cuenta con la solución del item anterior.

Solución

Problema 2.1.

Idea de algoritmo Primero daremos la idea del algoritmo para después presentar la solución.

El comportamiento del 2DFA $\mathcal{A}$ al leer la palabra ua lo podemos descomponer como leer primero la letra a y después leer u (por la derecha) si se mueve a la izquierda desde a. También sabemos que δ nos entrega el comportamiento en a y T_u el comportamiento en u.

Para calcular T_{ua}, necesitaremos considerar cómo $\mathcal{A}$ se comporta en la nueva posición $|u|+1$ al agregar la letra a. Si desde el estado $p \in Q$ se mueve a la derecha al leer a, esto es, $\delta(p, a) = (q, \rightarrow)$ para algún q, entonces inmediatamente sabemos que $T_{ua}(p) = q$ y estamos listos para p. Estos estados p lo recordaremos en un conjunto que llamaremos R.

En cambio, si desde p se mueve a la izquierda al leer a (esto es, $\delta(p, a) = (q, \leftarrow)$ para algún q), entonces debemos mirar que pasa en u usando la función T_u. Para lograr que todo el algoritmo tome tiempo $O(|Q|)$ vemos el comportamiento desde la configuración $(p, |u|+1)$ cuando se mueve a la izquierda (esto es, $\delta(p, a) = (q, \leftarrow)$ para algún q) y vuelve después a una configuración $(q', |u|+1)$ por primera vez. O sea, desde p leyendo a se mueve a la izquierda y después vuelve a la letra a al leer u. Esto lo podemos encontrar usando $T_u(q)$, o sea, desde p nos movemos a q y desde q volvermos al estado $T_u(q) = q'$. Este comportamiento lo almacenaremos en una tabla de Hash $E : Q \to Q$ como $E[p] = q'$. Veremos que el computo de E y R lo podemos hacer en tiempo $O(|Q|)$.

Para calcular $T_{ua}(p)$ lo que hacemos es usar el conjunto R y la tabla E. En particular, sabemos que si $E[p] = q$ y $q \in R$, entonces $T_{ua}(p) = T_{ua}(q)$. Este argumento lo podemos aplicar recursivamente hacia atras, o sea, si $E[p] = q$ y $T_{ua}(q)$ ya esta calculado, entonces $T_{ua}(p) = T_{ua}(q)$. Para hacer este calculo recursivo, lo que hacemos es invertir E, o sea, calcular $E^{-1} : Q \to 2^Q$ tal que $E^{-1}[q] = \{p \in Q \mid E[p] = q\}$. Entonces, partiendo desde R recorremos E^{-1} tal que si $T_{ua}[q]$ ya esta calculado y $p \in E^{-1}[q]$, entonces $T_{ua}(p) = T_{ua}(q)$. Por último, al terminar este proceso, las entradas $T_{ua}(p)$ que no están definidas es debido a que p entra en un loop desde $(p, |u|+1)$ y por lo tanto $T_{ua}(p) = \perp$.

La solución. En Algoritmo 1 mostramos los pasos para calcular T_{ua} usando la estrategia explicada anteriormente. El algoritmo recibe como input el 2DFA $\mathcal{A}$, la función T_u y la nueva letra $a \in \Sigma$, y calcula una tabla de Hash T_{ua} que representa la función $T_{ua} : Q \cup \{\circ\} \to Q \cup \{\bot\}$.

Algorithm 1: Calcular T_{ua}

Input: Un 2DFA $\mathcal{A} = (Q, \Sigma, \vdash, \dashv, \delta, q_0, F)$, una función $T_u : Q \cup \{\circ\} \to Q \cup \{\bot\}$ y una letra $a \in \Sigma$
Output: La función $T_{ua} : Q \cup \{\circ\} \to Q \cup \{\bot\}$

1 $T_{ua} \leftarrow \emptyset$ // Tabla de Hash $T_{ua} : Q \cup \{\circ\} \to Q \cup \{\bot\}$, inicialmente vacía.
2 $R \leftarrow \emptyset$ // R es un conjunto de estados, inicialmente vacío.
3 $E \leftarrow \emptyset$ // Tabla de Hash E: $Q \to Q$, inicialmente vacía.

4 **foreach** $p \in Q$ **do**
  // Al leer a, $\mathcal{A}$ se mueve a la derecha desde p.
5 **if** $\delta(p, a) = (q, \to)$ **then**
6 $T_{ua}[p] \leftarrow q$ // Definimos $T_{ua}[p] = q$
7 $R \leftarrow R \cup \{p\}$ // Recordamos que p es un estado tal que $\delta(p, a) = (q, \to)$ para algún q.
  // Al leer a, $\mathcal{A}$ se mueve a la izquierda desde p llegando a q.
8 **else if** $\delta(p, a) = (q, \leftarrow) \ \wedge \ T_{ua}[q] \neq \bot$ **then**
9 $q' \leftarrow T_u[q]$ // Obtiene el estado q' al que vuelve desde q.
10 $E[p] \leftarrow q'$ // Recordamos que desde $(p, |u| + 1)$ se mueve a la izquierda y vuelve a $(q', |u| + 1)$ por primera vez.

  // Invertimos la tabla E, construyendo $E^{-1} : Q \to 2^Q$ tal que $E^{-1}[q] = \{p \in Q \mid E(p) = q'\}$
11 $E^{-1} \leftarrow \emptyset$ // Tabla de Hash $E^{-1} : Q \to 2^Q$ inicialmente vacía.
12 **foreach** $p \in Q$ **do**
13 $q \leftarrow E[p]$
14 $E^{-1}[q] \leftarrow E^{-1}[q] \cup \{p\}$

15 $L \leftarrow R$ // Hacemos el recorrido inverso de E, partiendo desde los estados R. Notar que L es una cola.
16 **while** $L \neq \emptyset$ **do**
17 $q \leftarrow \text{pop}(L)$ // Saca un elemento de L
18 **foreach** $p \in E^{-1}[q]$ **do**
19 $\text{push}(L, p)$ // Agrega p a L
20 $T_{ua}[p] \leftarrow T_{ua}[q]$

  // Todo los estados donde $T_{ua}[p]$ no está definido significa que $\mathcal{A}$ desde $(p, |u| + 1)$ no sale a ninguna configuración $(q, |u| + 2)$.
21 **foreach** $p \in Q$ **do**
22 **if** $T_{ua}[p]$ *no esta definido* **then**
23 $T_{ua}[p] \leftarrow \bot$

  // Por último, vemos el caso de $T_{ua}(\circ)$ usando $T_u(\circ)$.
24 **if** $T_u(\circ) \neq \bot$ **then**
25 $T_{ua}(\circ) \leftarrow T_{ua}[T_u[\circ]]$
26 **else**
27 $T_{ua}(\circ) \leftarrow \bot$

28 **return** T_{ua}

Explicación del Algoritmo

- **Inicialización:** Se inicializan los conjuntos T_{ua}, R y E en la línea 1, 2 y 3 respectivamente. T_{ua} es una tabla de Hash (a completar) que representará la función $T_{ua} : Q \cup \{\circ\} \to Q \cup \{\bot\}$, R son los estados $p \in Q$ que al leer a se mueve inmediatamente a la derecha, esto es, $\delta(p, a) = (q, \to)$, y E es una tabla de Hash tal que $E[p] = q$ ssi $\mathcal{A}$ desde $(p, |u| + 1)$ se mueve a la izquierda y vuelve a $(q, |u| + 1)$ por primera vez.
- **Recorrer los estados posibles:** Se recorren cada estado p en el conjunto de estados Q en la línea 4. Si la transición de p al leer la letra a es $(q, \to)$, entonces se define directamente $T_{ua}[p] = q$ y p se almacena en R en la línea 6 y 7, respectivamente. Si la transición de p al leer la letra a es $(q, \leftarrow)$ y $T_{ua}[q] \neq \bot$,

entonces recordamos que desde $(p, |u|+1)$ se mueve a la izquierda y vuelve a $(T_{ua}[q], |u|+1)$ por primera vez, almacenando $E[p] = T_{ua}[q]$ en las líneas 9 y 10.

- **Invertir E:** Se invierte la tabla E en la línea 11-14. Esto se hace para poder simular el comportamiento "inverso" del autómata en la posición $|u|+1$ al leer la letra a.
- **Simulación inversa:** Se simula el comportamiento inverso del autómata en la posición $|u|+1$ al leer la letra a en las líneas 15 a 20. Si $T_{ua}[p] = \perp$, entonces se agrega $\perp$ a T_{ua} en la línea 23.
- **Simulación para $\circ$:** Si $T_{ua}(\circ) \neq \perp$, entonces se agrega $T_{ua}[T_u[\circ]]$ a T_{ua} en la línea 25. En otro caso, se agrega $\perp$ a T_{ua} en la línea 27.
- **Retorno:** Se retorna T_{ua} en la línea 28.

Complejidad La complejidad del algoritmo es $O(|\mathcal{A}|)$, donde $\mathcal{A}$ es el número de estados en el autómata (es decir, $|Q|$). Para ver esto, notar que en los loops de las líneas 4, 12 y 21 se recorre cada estado una sola vez, esto es, $O(|Q|)$. En cambio, el loop de la línea 16 toma tiempo $O(|Q|)$ debido a que no es posible agregar un estado q en L dos veces (linea 19). Si fuese así y q se agrega dos veces a L, entonces $\mathcal{A}$ tendría un loop desde la configuración $(q, |u|+1)$ a $(q, |u|+1)$ lo cuál no es posible ya que partimos desde los estados R que son todos de salida a la derecha (esto es, $\delta(p, a) = (q, \rightarrow)$ para todo $p \in R$). Por lo tanto, el loop de la linea 16 tomará tiempo $O(|Q|)$. Por último, para cada estado $p \in Q \cup \{\circ\}$ sólo realizamos un número constante de operaciones en cada loop como verificar $T_u(p)$, aplicar la función de transición δ o actualizar el valor de $T_{ua}(p)$. El argumento anterior garantiza que el algoritmo tomará tiempo lineal en el tamaño del autómata.

Distribución de puntaje

- 1 punto por recorrer cada estado correctamente y actualizar T_{ua} de acuerdo con las condiciones dadas.
- 1 punto por manejar correctamente la transición cuando autómata se mueve a la izquierda y $T_u[q]$ no es $\perp$.
- 1 punto por simular el comportamiento del autómata en la posición $|u|+1$ al leer la letra a.
- 1 punto por cubrir los casos en que $T_{ua}[p] = \perp$, $T_{ua}(\circ) \neq \perp$ y $T_{ua}(\circ) = \perp$.

Problema 2.2

Idea del algoritmo El autómata 2DFA puede moverse tanto a la izquierda como a la derecha sobre la cinta. Sin embargo, debido a la restricción de hacer sólo una pasada sobre la palabra, debemos simular el movimiento hacia la izquierda sin re-leer la palabra. Para ello, podemos mantener suficiente información en la memoria sobre los estados previos usando las funciones T_u y el algoritmo implementado en el item anterior.

El autómata comenzará en el estado inicial q_0 y procesará la palabra w desde el primer símbolo hasta el último, manteniendo la función T_u en la variable T. Al comienzo partirá con la función T_ϵ que podemos calcular usando δ. Después para cada letra a_i y la función T_u con $u = a_1 \dots a_i$, podemos calcular T_{ua_i} ocupando nuestro algoritmo del item anterior que tomará tiempo $O(|Q|)$.

Por último, al llegar al final de la palabra w habremos computado T_w. Usando $T_w[\circ]$ y el conjunto F podremos verificar si $\mathcal{A}$ acepta la palabra w o no.

La solución El algoritmo para evaluar un 2DFA sobre una palabra w es el siguiente:

Algorithm 2: Evaluar 2DFA en una sola pasada

Input: Un 2DFA $\mathcal{A} = (Q, \Sigma, \vdash, \dashv, \delta, q_0, F)$ y una palabra $w = a_1 a_2 \dots a_n$
Output: True si $w \in \mathcal{L}(\mathcal{A})$, False en otro caso.

```
1  T ← ∅ // Inicializar la variable T como vacía.
   // Calcular la función T_ε utilizando δ.
2  foreach p ∈ Q do
3  |  if δ(p, ⊢) = (q, →) then
4  |  |  T[p] ← q
5  |  else
6  |  |  T[p] ← ⊥
7  if δ(q_0, ⊢) = (q, →) then
8  |  T[∘] ← q
9  else
10 |  T[∘] ← ⊥
   // Simulamos A leyendo w letra por letra secuencialmente.
11 for i = 1 to n do
12 |  T ← Calcular T_ua(A, T, a_i)
   // Al llegar al final de la palabra T = T_w.
   // Si T[∘] = ⊥ entonces A nunca llega al final de w.
13 if T[∘] = ⊥ then
14 |  return False
   // Si T[∘] ≠ ⊥ entonces A llega al final de w pero debemos ver que llegué a un estado
      final ''rebotando'' en el símbolo final ⊣, a lo más |Q| veces.
15 p ← T[∘]
16 c ← |Q|
17 while c > 0 do
18 |  c ← c − 1
19 |  if p ∈ F then
20 |  |  return True
21 |  else if δ(p, ⊣) = (q, L) then
22 |  |  p ← T[q]
23 |  else
24 |  |  return False
25 return False
```

Explicación del Algoritmo

- **Inicialización:** Se inicializa el conjunto T en la línea 1. T es una tabla de Hash que representa una función T_u. Inicialmente asignamos T como vacío (línea 2) y después calculamos T_ϵ en T (lineas 2-10). Para esto vemos el comportamiento de $\mathcal{A}$ al leer el símbolo $\vdash$. Si $\delta(p, \vdash) = (q, \rightarrow)$, entonces $T_\epsilon[p] = q$. En cambio, si $\delta(p, \vdash) = (q, \leftarrow)$, entonces $T_\epsilon(p) = \perp$ (esto es, $\mathcal{A}$ cae "fuera"). Usando lo ya computado, en las líneas 7-10 calculamos $T_\epsilon[\circ]$.

- **Procesamiento de la palabra:** Se procesa la palabra w en las líneas 10 y 11. Para cada símbolo a_i en la palabra, se calcula la función de transición T_{ua}.
- **Verificación de aceptación:** Se verifica si el autómata está en un estado de aceptación en las líneas 13 a 24. Si el autómata está en un estado de aceptación, entonces se retorna True en la línea 20. Si el autómata necesita moverse hacia la izquierda, entonces se actualiza el estado del autómata en la línea 22. Si el autómata no está en un estado de aceptación y no necesita moverse hacia la izquierda, entonces se retorna False en la línea 24. Por último, si el ciclo while termina y aún no hemos retornado, rechazamos ya que significa que el autómata no ha aceptado la palabra debido a que se quedo pegado en un loop.

Complejidad:

- **Tiempo:** La complejidad del algoritmo es $O(|\mathcal{A}| \cdot |w|)$, donde $\mathcal{A}$ es el número de estados en el autómata (es decir, $|Q|$) y w es la longitud de la palabra. El algoritmo recorre cada símbolo en la palabra en el ciclo for. Además, para cada símbolo, se calcula la función de transición T_{ua} . Por lo tanto, el algoritmo toma tiempo lineal en el tamaño del autómata y la longitud de la palabra.
- **Espacio:** El algoritmo utiliza espacio adicional para mantener la tabla T y las variables auxiliares. Por lo tanto, el algoritmo toma espacio adicional $O(|\mathcal{A}|)$.

Distribución de puntaje

- 1 punto por inicializar correctamente la tabla T, $T[p]$ y $T[\circ]$.
- 1 punto por calcular correctamente la función de transición T_{ua} para cada símbolo en la palabra.
- 1 punto por verificar si el autómata está en un estado de aceptación y actualizar el estado del autómata si necesita moverse hacia la izquierda.
- 1 punto si el algoritmo tiene inicialización y verificación de aceptación correctas.

Evaluación y puntajes de la tarea

Cada item de cada pregunta se evaluará con un puntaje de 0, 1, 2, 3 o 4 puntos. Todas las preguntas tienen la misma ponderación en la nota final y cada item tiene la misma ponderación en cada pregunta.