Minimización de autómatas

Construcción de un AFDt con un <u>número de estados mínimo</u> que sea equivalente a un AFDt dado.

Definiciones previas:

- Estados accesibles: q0 es accesible
  q accesible $\Rightarrow \forall s \in \Sigma$, $\delta(q, s)$ es accesible
- Estados k-equivalentes o k-indistinguibles: $p \equiv_k q$

  $\forall x \in \Sigma^{\leq k} (\delta^*(p, x) \in F \leftrightarrow \delta^*(q, x) \in F)$
- Estados equivalentes o indistinguibles: $p \equiv q$

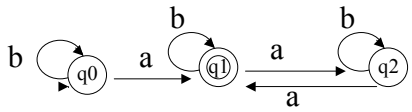
  $\forall k \; \forall x \in \Sigma^{\leq k} (\delta^*(p, x) \in F \leftrightarrow \delta^*(q, x) \in F)$, es decir,
  $\forall x \in \Sigma^{*} (\delta^*(p, x) \in F \leftrightarrow \delta^*(q, x) \in F)$

Minimización de autómatas

Construcción del AFDt **mínimo N** a partir del AFDt **M = (Q, Σ, δ, q_0, F)**

1) Eliminar estados inaccesibles
2) Determinar las clases de estados equivalentes:
   $p \equiv_0 q \Leftrightarrow (p \in F \leftrightarrow q \in F)$
   $p \equiv_{k+1} q \Leftrightarrow (p \equiv_k q \wedge \forall s \in \Sigma \; \delta(p, s) \equiv_k \delta(q, s))$
3) Construcción del AFD **N = (P, Σ, γ, p_0, G)** con
   P = Q/≡ siendo ≡ es la menor $\equiv_k$ tal que $\equiv_k$ coincide con $\equiv_{k+1}$
   $p_0 = [q_0]$
   $\gamma([p], s) = [\delta(p, s)]$
   $G = \{[p]: p \in F\}$

Ejemplo 1



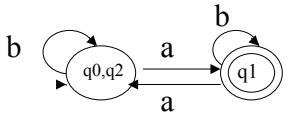
Etapa 0 : $q0 \equiv_0 q2$ (ambos $\notin$ F), pero q0 y q1 no son equivalentes
Clases a nivel 0: [q0,q2] [q1]

Etapa 1 : $q0 \equiv_1 q2$ porque $\delta(q0, a) \equiv_0 \delta(q2, a)$ y $\delta(q0, b) \equiv_0 \delta(q2, b)$

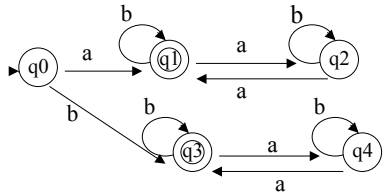
Clases a nivel 1: [q0,q2] [q1]

Por tanto $\equiv \; = \equiv_0$

AFD mínimo:



Ejemplo 2

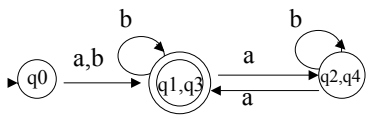


q0,q2	a	q1
	b	q3, q2 **no** equiv.
q0,q4	a	q1,q3
	b	q3, q4 **no** equiv.
q2,q4	a	q1,q3
	b	q2,q4
q1,q3	a	q2,q4
	b	q1,q3

Etapa 0 : [q0,q2,q4] [q1,q3]
Etapa 1 : [q0] [q2,q4] [q1,q3]
Etapa 2 : [q0] [q2,q4] [q1,q3]

Por tanto ≡ = $\equiv_1$

AFD mínimo :

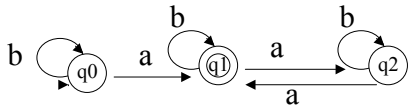


Método mediante tabla

- **Método:** marcar (X) en la tabla, los pares de estados que son **distinguibles**, haciendo un único recorrido de la tabla (de izquierda a derecha y cada columna de arriba a abajo).
- **Algoritmo de llenado de la tabla:**

1) Marcar todos los pares (p,q) ∈ F x (Q - F)
2) **Recorrer** la tabla y para cada (p,q) no marcado **hacer**
   si existe s∈Σ tal que (δ(p,s), δ(q,s)) está marcado
   entonces
      marcar (p,q)
      marcar recursivamente la lista de (p,q)
   sino
      añadir (p,q) a la lista de (δ(p,s), δ(q,s)) salvo que δ(p,s)=δ(q,s)

Tabla del ejemplo 1



q0	≡	simétrico	simétrico
q1	X	≡	simétrico
q2		X	≡
	q0	q1	q2

Etapa k=1

q0,q2	a	q1
	b	q0,q2

q0 y q2 son 1-equivalentes

[q0]=[q2]={q0,q2}

AFD mínimo:

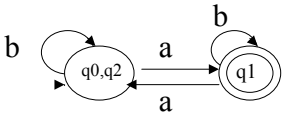
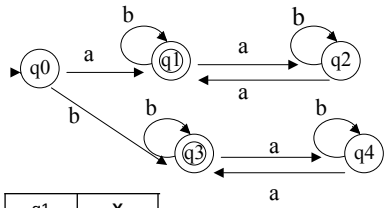


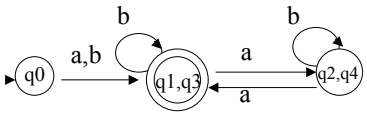
Tabla del ejemplo 2



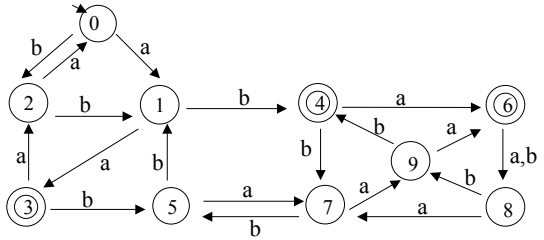
q1	X			
q2	X	X		
q3	X	(q2,q4)	X	
q4	X	X	(q1,q3)	X
	q0	q1	q2	q3

q0,q2	a	q1
	b	q3, q2 **marcado**
q0,q4	a	q1,q3
	b	q3, q4 **marcado**
q1,q3	a	q1,q3
A lista de **q2,q4**	b	q2,q4
q2,q4	a	q2,q4
A lista de **q1,q3**	b	q1,q3

AFD mínimo:



Ejemplo 3



Etapa 0 : [0,1,2,5,7,8,9] [3,4,6]
Etapa 1 : [0,2,5,7,8] [1,9] [3,6] [4]
Etapa 2 : [0,7] [2,5,8] [1,9] [3,6] [4]
Etapa 3 : se repite

Ejercicio:
- Dibujar el AFD mínimo
- ¿Tabla? Marcado recursivo

Minimización de autómatas

PROPOSICIÓN 9:
Dado un AFDt $M = (Q, \Sigma, \delta, q_0, F)$, la construcción anterior permite crear un AFDt $N = (Q/\equiv, \Sigma, \gamma, [q_0], G)$ mínimo y equivalente a M

DEMOSTRACIÓN:

- N es un AFD (γ está bien definida)

  $p \equiv q \Rightarrow \gamma([p], s) = \gamma([q], s) \quad \forall s \in \Sigma$
- N y M son equivalentes **L(N) = L(M)**

  Lema auxiliar: $\gamma^*([p], x) = [\delta^*(p, x)] \quad , \quad \forall x \in \Sigma^*$
- N es mínimo

Minimización de autómatas

N es mínimo:

Por Red.Absurdo: Supongamos M'= (Q', Σ, δ', q_0', F') con menos estados que N y tal que L(M') = L(N).

- M' y N son ambos AFDt con todos sus estados accesibles y M' tiene menos estados que N => tomando palabras que llegan a cada estado de N, dos de ellas (al menos) deben llegar al mismo estado en M' :

  $\exists\, u,v \in \Sigma^* : \ \delta'^*(q_0',u) = \delta'^*(q_0',v)$ y $\gamma^*([q_0],u) \neq \gamma^*([q_0],v)$
- Por construcción de N, dos estados diferentes son distinguibles:

  $\exists x \in \Sigma^*: \ \gamma^*([q_0],u.x) \in G \Leftrightarrow \gamma^*([q_0],v.x) \notin G$ y, por tanto,

  $u.x \in L(N) \Leftrightarrow v.x \notin L(N)$
- Pero $\delta'^*(q_0',u.x) = \delta'^*(q_0',v.x)$ implica que $u.x \in L(M') \Leftrightarrow v.x \in L(M')$
- Contradicción con L(M') = L(N).

Propiedades de cierre de los lenguajes regulares

Si L1 y L2 son lenguajes regulares

- L1 $\cup$ L2 es regular
- L1 $\bullet$ L2 es regular
- L1* es regular

  } Obvio a partir de expresiones regulares
- L1 $\cap$ L2 es regular
- $\overline{L1}$ es regular

La intersección de lenguajes regulares es un lenguaje regular

Sea M1 = (Q1, Σ, δ1, q_1, F1) un AFD con L(M1) = L1

Sea M2 = (Q2, Σ, δ2, q_2, F2) un AFD con L(M2) = L2

Construcción de M :

M = (Q, Σ, γ, q_0, F) con

Q = Q1 × Q2

q0 = <q_1, q_2>

F = F1 × F2

γ (<p, q>, s) = <δ1(p, s), δ2(q, s)>

- Lema: $\gamma^*(<p, q>, x) = <\delta 1^*(p, x), \delta 2^*(q, x)>$
- L(M) = L1 ∩ L2

El complementario de un leng. regular es un lenguaje regular

Sea M = $(Q, \Sigma, \delta, q_0, F)$ un AFDt con L(M) = L1

Construcción de N :

N = $(Q, \Sigma, \delta, q_0, G)$ con G = Q-F

- $L(N) = \overline{L(M)}$

$x \in L(N) \Leftrightarrow \delta^*(q_0, x) \in G \Leftrightarrow$

$\delta^*(q_0, x) \in Q\text{-}F \Leftrightarrow \delta^*(q_0, x) \notin F \Leftrightarrow$

$x \notin L(M) \Leftrightarrow x \notin L1$

Lenguajes no regulares

- Existen lenguajes que no son regulares y técnicas para demostrarlo (ver "El lema de bombeo" en la bibliografía dada)

Ejemplo: $L = \{ 0^n1^n : n \geq 0\}$ no es regular

Demostración:

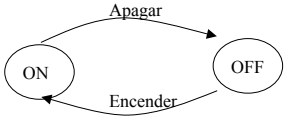
- Si L es regular, existe un AFDt $M = (Q, \Sigma, \delta, q_0, F)$ que lo reconoce.
- Consideramos el conjunto infinito $\{0^n : n \geq 0\}$
- M finito => deben existir 0^i y 0^j con $i \neq j$ tal que $\delta^*(q_0, 0^i) = \delta^*(q_0, 0^j)$
- Esto significa que $\delta^*(q_0, 0^i1^i) = \delta^*(q_0, 0^j1^i)$, pero por un lado $0^i1^i \in L$ y por otro $0^j1^i \notin L$. Llegamos a una contradicción.
- Por tanto no existe un AFDt M tal que L(M)=L .

Aplicaciones

- Interruptor de luz
- Control de máquinas de bebidas
- Analizadores/generadores de palabras
- Control de las tareas de un robot
- Búsqueda y sustitución de palabras
- Tratamiento de masas de textos
- Analizadores léxicos de compiladores y traductores
- etc.

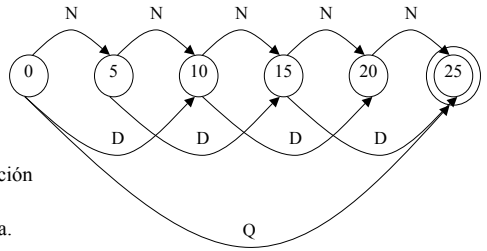
Aplicaciones

Interruptor de luz:



Máquina de bebidas:

1. Las bebidas cuestan 25 céntimos.
2. Monedas que admite la máquina:
   1. De cuarto, 25 céntimos (Q).
   2. Dimea, 10 céntimos (D).
   3. Nickel, 5 céntimos (N).
3. La máquina acepta cualquier combinación de monedas hasta 25 céntimos.
4. La máquina requiere la cantidad exacta.



Lenguaje natural

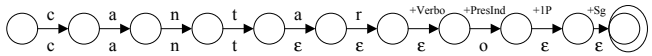
- Analizadores/generadores de palabras: Transductores

  Ejemplo: Uso de XFST (Xerox Finite State Transducer) en el análisis morfológico del castellano.

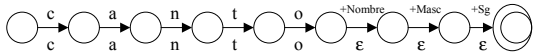
Análisis: analyze> canto

1. cantar+Verbo+PresInd+1P+Sg
2. canto+Nombre+Masc+Sg

1.análisis:



2.análisis:



Generación:

gener(ate)> cantar+Verbo+PresInd+1P+Pl

1. cantamos

Compilación

Analizadores léxicos

• Función de un compilador:

ENTRADA: un programa en ADA, C++, PERL, JAVA,…
SALIDA: código ejecutable en una máquina concreta

• Esa traducción se realiza en diferentes fases:

a) Reconocer entidades concretas del programa: palabras reservadas, identificadores, números, separadores, operadores, strings, …

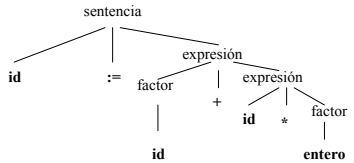
Ejemplo: posicion := inicial + velocidad * 60

Compilación

Ejemplo: posicion := inicial + velocidad * 60

id(posicion)	asignación(:=)
id(inicial)	operador(+)
id(velocidad)	operador(*)
entero(60)	

b) Reconocer la estructura de la secuencia de “tokens” del programa



c) A partir de dicha estructura, generar el código objeto

Compilación

El análisis léxico corresponde a la primera fase:

ENTRADA: una secuencia de caracteres (el programa)
SALIDA: una secuencia de “tokens” (las unidades del programa)

¿Cómo trabaja un analizador léxico?:

Lee la entrada carácter a carácter
Devuelve la secuencia que se adecúa con algún “token”
Para ello utiliza un autómata.

Compilación

En el análisis léxico hay que reconocer clases de palabras/ “tokens”. La definición de éstos se suele hacer a través de expresiones regulares:

números reales
("+"|-)? {digito}+ ("."{digito}+)? (E("+"|-)? {digito}+)?
identificadores
{letra} (_? ({letra}|{digito}))*
palabras reservadas
declare | function | if | then | else | elsif
operadores
:=|*|-|/

- Cada expresión regular se convierte en un ε-AFND
- Desde un estado inicial q0, y por medio de ε-transiciones, se unen todos los ε-AFNDs
- Se transforma en un AFD: el determinismo facilita el manejo

Generadores de analizadores léxicos

• Existen mecanismos que producen de forma automática analizadores léxicos: lex (UNIX), flex (GNU)...

• Crean el AFD asociado a una expresión regular y producen el código del analizador léxico (http://catalog.compilertools.net):

- Lenguaje C : lex, flex
- ADA: alex
- JAVA: jlex, jflex
- C++: lex++

Especificación lex

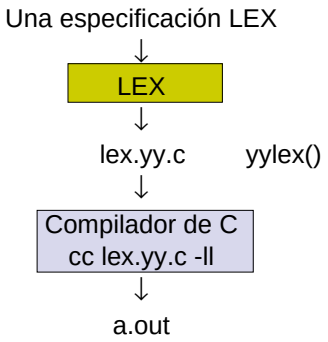
```
letra    [a-z A-Z]
digito   [0-9]
real     ("+"|-)? {digito}+ ("."{digito}+)? ( E("+"|-)? {digito}+ )?
identif  {letra} ( _? ({letra}|{digito}) )*
reser    (declare  | function  | if | then | else | elsif)
%%
reser    {es_palabra_reservada();}
real     {printf(“Real: %f\n",yytext);}
identif  {printf("Identificador: %s\n",yytext);}
.        ;
%%
void es_palabra_reservada()
{ printf(“Palabra reservada\n");}
```

Declaraciones

Reglas

Acciones

lex



- **Uso**

Cadena de entrada → a.out → Cadena de tokens