

L y NL

Ej 8:

## 8. Probar que la relación $\leq_L$ es transitiva.

Por definición de transitividad lo q' quiera probar es que:

$$(L \leq_L L' \text{ y } L' \leq_L L'') \Rightarrow L \leq_L L''$$

Entonces, por def. tengo q' existe  $f, g$  trabajo-L computables tales que:

$$x \in L \text{ sii } f(x) \in L' \quad (1)$$

$$y \in L' \text{ sii } g(y) \in L'' \quad (2)$$

Con esto veo que:

$$x \in L \stackrel{(1)}{\text{sii}} f(x) \in L' \stackrel{(2)}{\text{sii}} g(f(x)) \in L''$$

O sea:

$$x \in L \text{ sii } g(f(x)) \in L''$$

Ahora solo me queda ver que  $g \circ f$  sea trabajo-L computable (se ve en la prop. 25 de la teoría esta igual, voy a parrear la idea general nada más).

Tengo  $M_f$  y  $M_g$  + q'  $M_f(\langle x, i \rangle) = f(x)[i]$  y  $M_g(\langle x, i \rangle) = g(x)[i]$  y son ambas trabajo-L computables.

Luego,  $M_{g \circ f}$  va a funcionar como  $M_g$ , pero cada vez q' necesito el bit  $i$  de la entrada lo calculo con  $M_f(\langle x, i \rangle)$ , para esta tengo q' guardarme solamente el iterador de la entrada y salida de  $M_f(\langle x, i \rangle)$ .

Ej 9:

9. Sea  $\mathcal{L}$  el lenguaje de todas las expresiones con paréntesis bien formadas. Es decir,  $()$ ,  $(( ))$ ,  $(( ))()$   $\in \mathcal{L}$ , pero  $(( ))$ ,  $()()$   $\notin \mathcal{L}$ . Probar que  $\mathcal{L} \in L$ .

Algoritmo:  $\langle S \rangle$  <sup>string</sup>

$C := 0$

for ( $i = 0$ ;  $i < S$ ;  $i++$ ):

if ( $S[i] == '('$ ):  
 $C++$

if ( $S[i] == ')'$ ):  
 $C--$

if ( $C < 0$ ):  
ret false

ret ( $C == 0$ )

//  $|C| = |S|$ , es una especie de contador de paréntesis

// Itero por el string  $|i| = |S|$

} Si el char q' se lee es '(' se le suma a C, si es ')' se le resta, esta suma y resta llevaría cuenta de si ')' encontró su par '('.  
Si en algún momento C es negativo, significa q' se puso un ')' sin un '(' anteriormente. O sea, no es válido.

// Si todas las '(' tienen su par ')', devuelve true, pero hubo una cantidad equivalente de paréntesis (= cant. de restas y sumas)

Ej 10:

## 10. Probar que 2-COLOREO está en NL.

hint: Para q' un grafo sea 2-COLOREO no debe tener ningún ciclo de longitud impar. (Gracias al ayz q' me tiró el hint, tipazo).

Idea: Sé lo dicho en la hint. Como  $NL = coNL$ , puedo ver q'  $\rightarrow$  2-COLOREO está en NL y decir q' entonces 2-COLOREO está en NL.

Algoritmo:  $\langle G \rangle$  (de una m. q. det. N) ( $G = \langle V, E \rangle$  con  $V = \text{Vertex}$   $E = \text{Edges}$ )

```
For (i=0; i < |V|; i++): // i es un nodo (inicio del ciclo q' buscamos)  $O(\log |V|)$   
  C := 0 // C = Cant. de aristas ya recorridas (A lo sumo |V| para encontrar un ciclo.  $O(\log |V|)$ )  
  actual := i // actual lleva el nodo en el q' estoy parando del recorrido.  $O(\log |V|)$   
  m := 0 // m limita la longitud posible del camino  $O(\log |V|)$   
  while (m < |V|):  
    z := Genera un nodo  $\in \{0, \dots, |V|-1\}$  // Nodo random  $O(\log |V|)$   
    if ((actual, z)  $\in E$ ):  
      C++  
      if ((C es impar)  $\wedge$  (C > 1)  $\wedge$  (z == i)):  
        ret true  
      actual := z  
      m++  
  ret false
```

Entonces, el algoritmo q' computa la m. q' decide 2-COLOREO sería la q' compute N y luego niegue la salida de esta máquina, como  $NL = coNL$ , esta máquina es NL.

Ej 11:

11. Probar que los siguientes problemas son NL-completos.

- $SCC = \{\langle G \rangle : G \text{ es un grafo fuertemente conexo}\}$
- $NFA-NO-VACIO = \{\langle A \rangle : A \text{ es un autómata no determinístico que reconoce un lenguaje no vacío.}\}$

Digrafo donde se puede llegar desde un nodo a cualquier otro

a) SCC es NL-completo.

Primero veo que  $SCC \in NL$ , la idea del algoritmo es modificar un poco el de PATH para q' se convierta en el problema q' quiero computar.

Algoritmo:  $\langle G \rangle$

res := true

For( $i=0; i < |V|; i++$ ):  
For( $j=0; j < |V|; j++$ ):

if( $i \neq j$ ):

existeC := false  
actual := i

For( $m=0; m < |V|; m++$ ):

z := Genero un nodo  $\in \{0, \dots, |V|-1\}$

if( $(actual, z) \in E$ ):

if( $actual == j$ ):

existeC := True

actual := z

res := res && existeC

ret res

Es un o.s.L.

// Resultado Final de si todos los nodos

$O(1)$

} Itero por todos los nodos viendo si hay un camino entre ellos  $O(2 \log |V|)$

// existe Camino entre estos 2 nodos? \*  $O(\log |V|)$

$O(1)$

Algoritmo de PATH básicamente

// Uno el resultado de \* con un and, así condiciona a q' todos los caminos deban existir.

Con este algoritmo se ve q' SCC  $\in$  NL. Otra forma hubiese sido aplicar algo similar a lo hecho en el punto anterior, demostrando q'  $\neg$  SCC  $\in$  NL.

Ahora tengo que ver que SCC  $\in$  NL-hard. Para esto, puedo usar PATH (q' se q' es NL-hard) para la reducción:

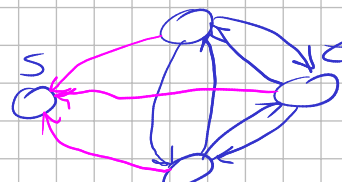
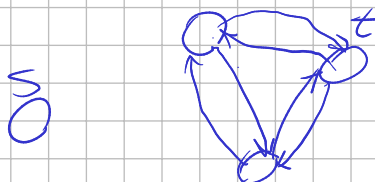
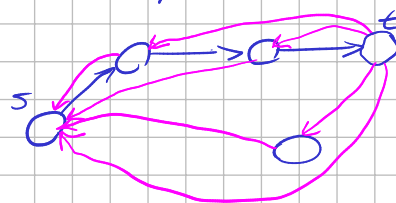
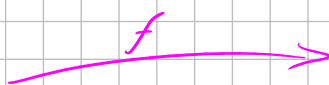
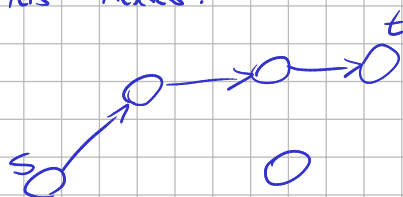
$x \in \text{PATH}$  sii  $f(x) \in \text{SCC}$  con  $f$  trabajo-L computable  
PATH  $\leq_L$  SCC

• Defino  $f$  de la siguiente forma:  $\langle G, s, t \rangle$  es la entrada

$f$  toma a un digrafo  $G$  y construye  $G'$  tal que:

- Por cada nodo de  $V_G$  lo copio en  $V_{G'}$  y cada arista de  $E_G$  también la copio a  $E_{G'}$  y además hago q' cada nodo de  $V_G$  se conecte a  $s$  y que  $t$  se conecte a todos los nodos.

Visuali



Algoritmo de  $f$ :

Agrego a la salida los nodos y las aristas de  $G$   $O(1)$

for( $i:=0$ ;  $i < |V|$ ;  $i++$ ):  $O(\log |V|)$

if( $i \neq s$ ):

if( $i == s$ ):

for( $j:=0$ ;  $j < |V|$ ;  $j++$ ):  $O(\log |V|)$

if( $(j \neq t) \wedge ((t, j) \notin E_G)$ ):

Agrego  $(t, j)$  a la salida

elif( $(i, s) \notin E_G$ ):

Agrego  $(i, s)$  a la salida

Agrego las aristas desde  $t$  a todas las otras nodos

Agrego las aristas de todos los nodos a  $s$ .

Finalizo la salida.

• Por qué funciona esta  $f$ ?

→ Bueno, si yo tengo un camino de  $s$  a  $t$ , entonces voy a tener en  $G'$  un camino de cualquier nodo a  $s$  y de  $s$  a  $t$ , el cual tiene una arista directa a cualquier otro nodo.

Si no existe un PATH de  $s$  a  $t$ , entonces en  $G'$  sigue sin haber un camino de  $s$  a  $t$ , ya q' no se agregan aristas de  $s$  a ningún nuevo nodo y al resto de nodos ( $\neq t$ ) se le agregan aristas hacia  $s$ , o sea, si en  $G$  no se llega a  $t$ , entonces en  $G'$  tampoco por lo cual no es fuertemente conexo.

(F) La vuelta queda medio trivial pq' depende de  $G$  para construir  $G'$ , por lo cual, por lo dicho en la ida, solo se tiene un  $G'$  fuertemente conexo si existe un PATH de  $s$  a  $t$ , si no existe en  $G'$ , en  $G$  tampoco.

6)  $\overbrace{\text{NFA-no-vacio}}^{\pi}$  es NL-completo.

Primero veo q'  $\pi$  pertenece a NL, para lo cual hago el siguiente algoritmo:

Idea: Si el autómata analiza una palabra con más de  $|Q|$  símbolos significa q' pasó por algún ciclo, por lo q' lo podría haber evitado y obtenido una palabra más chica.

Entonces hay al menos 1 palabra de a lo sumo  $|Q|$  símbolos q' pertenece al lenguaje.

Algoritmo:  $\langle Q, q_0, \Sigma, F, \delta \rangle$

actual :=  $q_0$   $O(\log(n))$

for( $m:=0$ ;  $m < |Q|$ ;  $m++$ ):  $O(\log |Q|)$

$s :=$  Inventa un símbolo  $\in \Sigma$

$q_{\text{next}} :=$  Inventa un estado  $\in Q$

if(existe  $\delta(\text{actual}, q_{\text{next}}, s)$ ):

$O(\log |\Sigma|)$  y  $O(\log |Q|)$ .

Busco un camino a algún estado final desde el inicial de longitud de a lo sumo  $|\Sigma|$

if ( $q_{next} \in F$ ):

ret true

actual :=  $q_{next}$

ret false

Luego, debo ver  $q' \in \Pi$  es NL-hard. Por lo cual voy a querer reducir desde  $PATH \leq \Pi$ , o sea  $PATH \leq \Pi$

$x \in PATH$  sii  $f(x) \in \Pi$  con  $f$  trabajo-L computable.

Tomando una instancia  $\langle G, s, t \rangle$  de  $PATH$ , lo q' haría sería generar un autómata  $M = \langle Q, q_0, \Sigma, F, \delta \rangle$  de la siguiente manera:

- $Q :=$  nodos de  $G$
- $q_0 := s$
- $\Sigma := \{1\}$
- $F := \{t\} = \{q_f\}$
- $\delta :=$  Si existe la arista  $(a, b)$  en  $G$ , existe la transición  $\delta(a, b, 1)$ .

Esta  $f$  es trabajo-L computable pq' va en orden 1 a 1 leyendo las cosas en la cinta de salida. Luego demuestra:

$x \in PATH$  sii  $f(x) \in \Pi$   
① ②

$\Rightarrow$ ) Si ① es verdadero, entonces  $f(x) = M$  va a tener una sucesión de  $\delta$  de  $q_0 \rightarrow q_f$  ya q' las transiciones se corresponden con las aristas y los estados con los nodos y  $q_0 = s$ ,  $q_f = t$ .

Si ① es falso, entonces desde el  $q_0$  de  $M$  no hay camino a  $q_f$ , por lo cual el lenguaje q' decide  $\emptyset$ .

$\Leftarrow$ ) Es muy análoga a la ida ( $\Rightarrow$ ), ya que por def. si hay un camino de  $q_0 \rightarrow q_f$ , es pq' habrá un camino de  $s \rightarrow t$  en  $G$ .

✓

Ej 12:

## 12. Probar que 2-SAT $\in$ NL.

hint:  $(s \vee t) \equiv (s \Rightarrow t) \equiv (\neg t \Rightarrow s)$  (gracias a otra ayz !!)

Vamos medio desprecito q' es una demo raro...

Primeramente si q' puedo traducir a cualquier fórmula de 2-SAT a otra fórmula con el estilo de la hint.

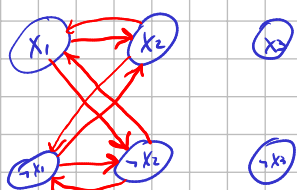
Luego, ya con la fórmula de esta manera puedo convertirlo en un grafo, de manera que cada nodo sean las variables de la fórmula y sus negación.

Visual:

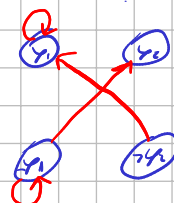


Luego, le agrego aristas de manera q' se correspondan con las implica de la fórmula

Visual:  $\varphi$ :



$\psi$ :



Si solo pongo q'  
 $s \vee t \equiv \neg s \Rightarrow t$  y  
 no q'  $s \vee t \equiv (\neg s \Rightarrow t) \wedge (\neg t \Rightarrow s)$   
 Se rompe.

contradicción

$$\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2) = (\neg x_1 \Rightarrow x_2) \wedge (x_1 \Rightarrow x_2) \wedge (\neg x_1 \Rightarrow \neg x_2) \wedge (x_1 \Rightarrow \neg x_2) \\ \wedge (\neg x_2 \Rightarrow x_1) \wedge (\neg x_2 \Rightarrow \neg x_1) \wedge (x_2 \Rightarrow x_1) \wedge (x_2 \Rightarrow \neg x_1)$$

fórmula satisficible

$$\psi = (y_1 \vee \neg y_1) \wedge (y_2 \vee \neg y_2) = (\neg y_1 \Rightarrow \neg y_1) \wedge (\neg y_2 \Rightarrow \neg y_1) \wedge (y_1 \Rightarrow y_1) \wedge (\neg y_1 \Rightarrow y_2)$$

Por último, voy a ver q' si la fórmula es satisficible, entonces no puedo tener un ciclo q' comience y termine en el  $x_i$  y pase por  $\neg x_i$ .

Obviamente tengo q' demostrar q' esta la resuelvo con un algoritmo en NL (esto se ve fácil pq ya tengo PATH, solo habría q' hacerle algunas modificaciones (q' se fija si pasa por  $\neg x_i$  desde  $x_i$  a  $x_i$ ) para conseguir lo que quiero).

También debo justificar que  $x \in$  2-SAT sii algoritmo aplicado a  $x = 1$

• Algoritmo NL para hacer la vista:

Sería básicamente buscar un ciclo de  $x_i$  que pase por  $\neg x_i$ , si esto no sucede significa q' la fórmula es satisficible. Por lo visto varias veces el algoritmo de Reachability es NL.

PATH( $G, x_i, x_i$ )

Adentro chequea q' pase por  $\neg x_i$ , si pasa retorna true

Esto es el complemento de lo q' quiero, pero como NL = coNL puedo simplemente después de computarlo negar la salida.

- ¿Qué significa q' haya un ciclo?

C

Si hay un ciclo en el grafo q' cumple q' contiene a  $x_i$  y  $\neg x_i$ , esto significa q' llega un momento donde se puede decir q'  $x_i \Rightarrow \neg x_i$  y  $\neg x_i \Rightarrow x_i$ , y esto es siempre falso.

- Pero te dan una fórmula, no un grafo. ¿cómo podrías hacer lo anterior en logspace?

Esto se soluciona concluyendo q' la función q' transforma  $\varphi$  a  $G$  es trabajable computable, o sea, podría calcularlo on-the-fly, o sea a medida de q' sea necesario lo calculo.

✓