L y NL

Ej 8:

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

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

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

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

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

$y \in \mathcal{L}'$ sii $g(x) \in \mathcal{L}''$ ②

Con esto veo que:

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

O sea:

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

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

Tengo M_f y M_g tq' $M_f(\langle x, i\rangle) = f(x)[i]$ 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' necesita el bit i de la entrada lo calculo con $M_f(\langle x, i\rangle)$, para esto 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: ⟨S⟩ (string)

```
c := 0                          // |c| = |s|, es una especie de contador de paréntesis
for (i := 0; i < s; i++):       // Itero por el string |i| = |s|
  if (S[i] == '('):             Si el char q' se lee es '(' se le suma a c, si es
    c++                         ')' se le resta, esta suma y resta llevaría cuenta de si ')'
                                encontró su par '('.
  if (S[i] == ')'):             Si en algún momento c es negativo, significa q'
    c--                         se puso un ')' sin un '(' anteriormente. O sea, no es
                                válida.
    if (c < 0):
      ret false
ret (c == 0)                    // Si todos los '(' tienen su par ')', devuelve true, pues
                                hubo una cantidad equivalente de paréntesis (= cant. de restas y sumas)
```

y

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' ¬2-COLOREO está en NL y decir q' entonces 2-COLOREO está en NL.

Algoritmo ⟨G⟩ (de una máq. no det. N) (G = ⟨V,E⟩ con V = Vertex E = 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
  actual := i                        un ciclo. O(log |V|)
  m:=0                            // actual lleva el nodo en el q' estoy parado del recorrido. O(log |V|)
                                  // m limita la longitud posible del camino O(log |V|)
  while (m <|V|):
    z:= Genero un nodo ∈{0, ..., |V|-1}     // Nodo random  O((log|V|)
    if((actual, z) ∈E ):
      C++
      if ((C es impar) y (C >1) y (z == i)):      } Si llego a un ciclo impar co más de
                                                    1 arista devuelvo true.
        ret true
      actual := z                 // Si no, sigo alargando el recorrido.
    m ++
ret false
```

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

γ

Ej 11:

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

- SCC = {⟨G⟩ : G es un grafo fuertemente conexo} (Digrafo donde se puede llegar desde un nodo a cualquier otro)
- NFA-NO-VACIO = {⟨A⟩ : A es un autómata no determinístico que reconoce un lenguaje no vacío.}

a) SCC es NL-completo.

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

Algoritmo: <G>

Es un o ó 1.

res := true // Resultado final de si todos los nodos ↑ O(1)

For(i=0; i< |V|; i++): } Itero por todos los nodos viendo si hay un camino
  For(j=0; j<|V|; j++): entre ellos O(2 log |V|)

  if(i ≠ j):

    existeC := false // ¿existe camino entre estos 2 nodos?* O(1)
    actual := i O(log |V|)

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

      z:= Genero un nodo ∈ {0, ..., |V|-1} O(log |V|)

      if((actual, z) ∈ E):

        if(actual == j):

          existeC := True

        actual := z

    res := res && existeC // Uno el resultado de * con un and, así condiciono a q' todos los caminos deban existir.

ret res

Algoritmo de PATH básicamente

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

Ahora tengo que ver que SCC ∈ NL-hard.

Después lo sigo pasando (lo tengo en la carpeta, pero quiero hacer otros ejercicios. ^^

Ej 12:

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

Hint: $(s \vee t) \equiv (s \Rightarrow t) \wedge (\neg t \Rightarrow s)$ (gracias a otra ayu ^.^)

Vamos medio despacito q' es una demo rara...

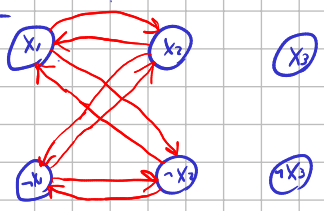
Primeramente sé q' puedo traducir a cualquier fórmula de 2-SAT a otra fórmula con el estilo de la hint.

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.

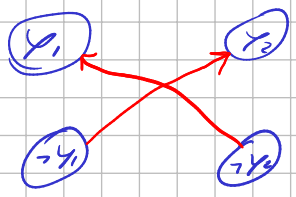
Visual:

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

Visual: φ:



Ψ:



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

$$\Psi = (y_1 \vee y_2) = (\neg y_1 \Rightarrow y_2) \vee (\neg y_2 \Rightarrow y_1)$$

Por última, voy a ver q' si la fórmula es satisfacible, 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 pequeñas modificaciones (q' se fije si paso 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 $x = 1$

Para preguntar: Cómo hago para q' el utilizar un grafo creado esté en logspace? Debería hacer una cinta q' de el grafo de a bits?