Modelos Avanzados de Computación

Relación de Ejercicios 4

5. Sea $L = \{wcw : w \in \{0,1\}^*\}$, Demostrar que L se puede reconocer en espacio $\log(n)$.

Usamos un algoritmo similar al de reconocer si una palabra es un palíndromo, que también es O(log(n)).
En la primera cinta tenemos la entrada.
En la segunda cinta guardamos la posición que estamos comprobando. (N2)
En la tercera cinta guardamos el número de la casilla por la que vamos. (N3)

Ponemos un 1 en las dos cintas inferiores (N2 = N3 = 1).

Repetir:
- Poner N3=1, desplazarse al principio de la primera cinta.
- Repetir hasta N2=N3: (Pongo el cabezal de la primera cinta en la posición que indique N2)
  - Incrementar N3 en 1, mover a la derecha en la primera cinta.
- Memorizar el símbolo de la primera cinta.
- ~~Si el símbolo memorizado es c: **Aceptar** (Se acabó w)~~
- ~~En otro caso:~~
  - Poner N3=1, desplazarse a la derecha en la primera cinta hasta c
  - Repetir hasta N2=N3:
    - (Pongo el cabezal de la primera cinta en la posición que indique N2, tras c)
    - Incrementar N3 en 1, mover a la derecha en la primera cinta.
  - ~~Si el símbolo en la primera cinta es distinto al memorizado: **Rechazar**~~
  - Incrementar N2 en 1 (Para comprobar siguiente símbolo)

0	1	c	0	1	#	Entrada	
1	1	#	#	#	Posición que se está comprobando (binario)	N2	
1	1	#	#	#	Contador para encontrar posiciones (binario)	N3	

Si el símbolo memorizado es "c" y el símbolo de la 1ª cinta es #: ACEPTAR
Si los dos símbolos son distintos: RECHAZA

Nunca escribimos en la cinta de entrada ⇒ No la contamos
Las dos cintas auxiliares contienen números en binario correspondientes a posiciones de la entrada, luego necesitan log(n) casillas, donde n es la longitud de w, menos de la mitad de la longitud de la entrada.

6. Demostrar que si L esta en $\mathbf{P}$, entonces L^* también está en $\mathbf{P}$.

Para demostrar que si un lenguaje L está en P, entonces L* también está en P, debemos mostrar cómo construir un algoritmo determinístico de tiempo polinómico para L*.

Recordemos que L* es el conjunto de todas las cadenas de símbolos que se pueden obtener concatenando cualquier número finito de cadenas de L. Formalmente, L* = {w1w2...wn | n ≥ 0, wi ∈ L para i = 1, 2, ..., n}.

Para construir un algoritmo determinístico de tiempo polinómico para L*, podemos utilizar el siguiente enfoque. Supongamos que tenemos un algoritmo determinístico de tiempo polinómico para L, es decir, un algoritmo que decide si una cadena w pertenece a L o no en tiempo polinómico. Entonces, podemos construir un algoritmo determinístico de tiempo polinómico para L* de la siguiente manera:

1. Dada una cadena w en L*, podemos escribirla como w = w1w2...wn, donde cada wi es una cadena en L.
2. Para cada wi, aplicamos el algoritmo de decisión de L para verificar si wi está en L o no. Si en algún momento encontramos una cadena wi que no está en L, detenemos la ejecución y rechazamos w. De lo contrario, seguimos procesando las siguientes cadenas wi hasta llegar al final de la cadena w.
3. Si hemos procesado todas las cadenas wi y ninguna ha sido rechazada, entonces aceptamos w.

La clave para que este algoritmo sea de tiempo polinómico es que el número de cadenas wi que debemos procesar es finito y está acotado por el número de símbolos en w. Por lo tanto, el tiempo de ejecución del algoritmo es proporcional al número de símbolos en w, lo que es polinómico.

En resumen, si L está en P, podemos construir un algoritmo determinístico de tiempo polinómico para L* utilizando el algoritmo de decisión de L. Por lo tanto, L* también está en P.

7. Demostrar que si L esta en **NP**, entonces L^* también está en **NP**.

Adivinar = Elegir de forma no determinista

Para demostrar que si un lenguaje L está en NP, entonces L* también está en NP, debemos mostrar cómo construir un algoritmo no determinístico de tiempo polinómico para L*.

Recordemos que L* es el conjunto de todas las cadenas de símbolos que se pueden obtener concatenando cualquier número finito de cadenas de L. Formalmente, L* = {w1w2...wn | n ≥ 0, wi ∈ L para i = 1, 2, ..., n}.

Para construir un algoritmo no determinístico de tiempo polinómico para L*, podemos utilizar el siguiente enfoque. Supongamos que tenemos un algoritmo no determinístico de tiempo polinómico para L, es decir, un algoritmo que verifica si una cadena w pertenece a L en tiempo polinómico. Entonces, podemos construir un algoritmo no determinístico de tiempo polinómico para L* de la siguiente manera:

1. Dada una cadena w en L*, podemos escribirla como w = w1w2...wn, donde cada wi es una cadena en L.
2. A continuación, adivinamos una partición de la cadena w en k subcadenas w1', w2', ..., wk', donde cada wj' es una subcadena de wi. Es decir, adivinamos dónde están las divisiones entre las subcadenas de L que componen w.
3. Para cada wj', verificamos si wj' está en L utilizando el algoritmo no determinístico de tiempo polinómico para L. Si en algún momento encontramos una cadena wj' que no está en L, rechazamos la adivinanza actual de partición y generamos una nueva adivinanza. De lo contrario, seguimos procesando las siguientes cadenas wj' hasta llegar al final de la cadena w.
4. Si hemos procesado todas las subcadenas wj' y todas están en L, entonces aceptamos la adivinanza actual de partición y aceptamos w. Si no, rechazamos la adivinanza actual de partición y generamos una nueva adivinanza.

La clave para que este algoritmo sea de tiempo polinómico es que el número de subcadenas wj' que debemos procesar es finito y está acotado por el número de símbolos en w. Por lo tanto, el tiempo de ejecución del algoritmo es proporcional al número de símbolos en w, lo que es polinómico.

En resumen, si L está en NP, podemos construir un algoritmo no determinístico de tiempo polinómico para L* utilizando el algoritmo no determinístico de tiempo polinómico para L. Por lo tanto, L* también está en NP.

Construyo una máquina de Turing que coloca o no (de forma no determinista) separadores entre cada dos símbolos de la palabra de entrada (la copia abajo con separadores). Acabo de construir una partición de la palabra de entrada arbitraria. Como para eso hay que recorrerla y copiarla, esto se hace en O(n).
Tras esto tengo que ir recorriendo otra vez la palabra de entrada (con separadores), ya con un número finito de separadores puestos, y comprobando que cada porción pertenece a L. Ambas cosas necesitan espacio polinómico para realizarse, recorrer la palabra con separadores es O(n) (como mucho habrá n+1 separadores) y determinar si cada porción pertenece a L se hace en O(k) donde k es la longitud de la porción, que también es O(n). Por tanto comprobar si la entrada está en L^* se realiza en tiempo polinómico con una máquina no determinista.

8. Demostrar que **NP** es cerrada para la unión y la intersección.

Adivinar = Elegir de forma no determinista

Para demostrar que la clase de complejidad NP es cerrada para la unión y la intersección, debemos demostrar que si tenemos dos lenguajes L1 y L2 en NP, entonces la unión y la intersección de estos lenguajes también están en NP.

Empezamos con la unión de L1 y L2. Para demostrar que L1 ∪ L2 está en NP, podemos utilizar el siguiente algoritmo no determinístico de tiempo polinómico:

1. Dada una cadena w, adivinamos si w pertenece a L1 o a L2.
2. Si adivinamos que w pertenece a L1, verificamos si w está en L1 utilizando un algoritmo no determinístico de tiempo polinómico para L1. Si lo está, aceptamos w; de lo contrario, rechazamos.
3. Si adivinamos que w pertenece a L2, hacemos lo mismo para L2.
4. Si en ambos casos rechazamos, entonces rechazamos w.

Este algoritmo funciona porque el tiempo de ejecución es proporcional a la longitud de w, lo que es polinómico. Además, si w está en L1 ∪ L2, entonces siempre existe una adivinanza válida que nos permita aceptar w.

Ahora pasamos a la intersección de L1 y L2. Para demostrar que L1 ∩ L2 está en NP, podemos utilizar un argumento similar. Podemos utilizar el siguiente algoritmo no determinístico de tiempo polinómico:

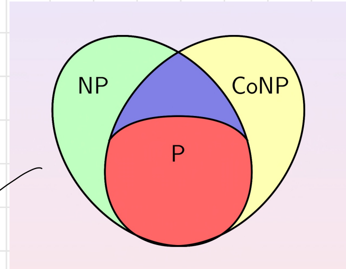
1. Dada una cadena w, adivinamos si w pertenece a L1 y a L2.
2. Si adivinamos que w pertenece a L1, verificamos si w está en L1 utilizando un algoritmo no determinístico de tiempo polinómico para L1. Si lo está, continuamos con el siguiente paso; de lo contrario, rechazamos.
3. Si adivinamos que w pertenece a L2, hacemos lo mismo para L2.
4. Si en ambos casos aceptamos, entonces aceptamos w; de lo contrario, rechazamos.

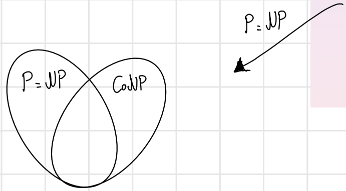
Este algoritmo también funciona porque el tiempo de ejecución es proporcional a la longitud de w, lo que es polinómico. Además, si w está en L1 ∩ L2, entonces siempre existe una adivinanza válida que nos permita aceptar w.

En resumen, hemos demostrado que la clase de complejidad NP es cerrada para la unión y la intersección. Esto significa que si tenemos dos lenguajes L1 y L2 en NP, entonces L1 ∪ L2 y L1 ∩ L2 también están en NP.

9. Demostrar que si $\mathbf{NP} \neq \mathbf{CoNP}$ entonces $\mathbf{P} \neq \mathbf{NP}$

Por contrarecíproco. Supongo que P = NP y vamos a probar que NP = CoNP. Como P = NP, tendremos que CoP = CoNP y sabemos que P = CoP, luego NP = CoNP.





10. Supongamos que una entrada es una palabra de paréntesis. Demostrar que determinar si están emparejados y anidados correctamente está en $\mathbf{L}$. Lo están (()()) y ((())), pero no lo están)()(ni (()))(.

Diseñamos el siguiente algoritmo:
Tenemos dos cintas, la cinta de entrada y una segunda cinta con un contador.
Vamos leyendo la palabra y por cada paréntesis (incrementamos en uno el contador en la segunda cinta y por cada paréntesis) lo decrementamos, pero si el contador ya era 0, la palabra no pertenece al lenguaje.
La palabra pertenece al lenguaje si la segunda cinta termina con un 0 cuando se termine de leer la entrada.
La palabra tendrá como mucho n paréntesis (y para escribir n en binario necesito $\log_2(n)$ casillas. Por tanto, el algoritmo tiene una complejidad de $O(\log_2(n))$, luego pertenece a L.