IIC2213 — Lógica para Ciencia de la Computación

Notas Semana 9

1. La clase NP o NPTIME

1.1. Definiciones

Partimos por definir la noción de $t_M(w)$ para máquinas no-deterministas: el tiempo que se demora M en procesar inputs de tamaño w.

Para una máquina de turing no-determinista M sobre un alfabeto A que para en todas las entradas y una palabra w que es aceptada por M, definimos $p^*_M(w)$ como la cantidad de pasos que ejecuta M con input w hasta que acepta w, en la ejecución más corta dentro de todas las ejecuciones que aceptan a w (es decir, todas las que con input w se detienen en un estado final).

Nota que sólo definimos $p^*_M(w)$ para las palabras que acepta M.

El *peor tiempo* para M con inputs de tamaño n se escribe como $t_M(n)$, y se define esta vez como

$$t_M(n) = \text{máx}\big\{\{n\} \cup \{p^*_M(w) \mid w \in \mathbf{A}^*, |w| = n \text{ y } w \in L(M)\}\big\}.$$

Nota que ahora definimos $t_M(n)$ como el máximo entre n y $p^*_M(w)$ para todas las palabras w de largo n. La razón por la que hacemos esto (en vez de solo definir como el máximo de todos los $p^*_M(w)$ para $w \in L(M)$) es por que nos gusta que $T_M(n)$ sea continua, y para una máquina M puede que para ciertos n la máquina M no acepte a ningún string de largo n, en cuyo caso $p^*_M(w)$ no estaría definido para ningún w de ese largo.

Definición. Sea $t : \mathbb{N} \to \mathbb{N}$ una función sobre los naturales. Decimos que un lenguaje L puede ser aceptado en tiempo t por una máquina de turing no-determinista M si (1) $L = L(M)$ y (2) $t_M(n)$ es $O(t)$; en otras palabras, existe un entero n_0 y una constante real c tal que $t_M(n) \leq ct(n)$ para cada $n \geq n_0$.

Definición. Para una función t, denotamos por $\text{NTIME}(t)$ al conjunto de todos los lenguajes que pueden ser aceptados en tiempo t por una máquina no-determinista.

Finalmente podemos definir la clase NPTIME:

- $\text{NPTIME} = \bigcup_{k\in\mathbb{N}} \text{NTIME}(n^k)$,

1.2. Relaciones con otras clases

Teorema.

- $\text{PTIME} \subseteq \text{NPTIME}$

- NPTIME $\subseteq$ EXPTIME

Demostración. El primer ítem sale automático del hecho que toda máquina no-determinista es también una máquina determinista.

Para el segundo ítem necesitamos refinar la transformación que hicimos para pasar de una máquina determinista a una no-determinista. Aunque los detalles se escapan un poco de este curso, podemos dar una intuición. La demostración sale del siguiente Lema. ¿Puedes ver por qué?

Lema. Sea M una máquina no-determinista sobre un alfabeto A y que para en todas las entradas, y tal que $t_M(n)$ es $O(n^k)$ para algún entero k fijo. Entonces, es posible construir una máquina de turing determinista M_D tal que (1) $L(M) = L(M_D)$, es decir que ambas máquinas aceptan el mismo lenguaje, y (2) $T_{M_D}(n)$ es $O(2^{(n^\ell)})$ para algún entero ℓ.

Puedes revisar tus notas sobre la transformación de una máquina determinista en una no-determinista, y ver como sale el lema de esta transformación.

1.3. SAT y NP

Recordemos el lenguaje de CNF-SAT:

$$\text{CNF-SAT} = \{w \in \{0,1\}^* \mid w = C(\varphi), \text{ para } \varphi \text{ fórmula de } L(P), \text{ en CNF, y tal que } \varphi \text{ es satisfacible}\}$$

La semana pasada vimos que CNF-SAT estaba en EXPTIME, y que no sabíamos si estaba en PTIME. Ahora mostramos que también pertenece a la clase NPTIME!

Proposicion. El lenguaje CNF-SAT pertenece a NPTIME.

Demostración. Recuerda que mostramos que CNF-SAT pertenece a EXPTIME, y la idea era que podíamos iterar sobre todas las posibles valuaciones para la fórmula del input, hasta que alguna de esas valuaciones la satisfaga (luego de eso, ya sabemos que también tenemos otra máquina para resolver CNF-EVAL, el problema de saber si una valuación satisface a una fórmula, en tiempo polinomial).

Para mostrar que CNF-SAT está en NPTIME, vamos a hacer lo mismo, pero sustituyendo esta iteración por no-determinismo: nuestra máquina va a "adivinar" la valuación correcta, si esta existe. Para eso, vamos por parte.

Observación. Podemos construir, dado un número n escrito como la palabra 1^n, una máquina de turing no-determinista M_b que tenga 2^n posibles ejecuciones, todas finales, pero tal que en cada una de esas ejecuciones la cinta de trabajo tiene, al final, un string binario de n caracteres distinto.

La máquina es simple: tiene un estado q_0, y una única transición

$$\delta(q_0, 1) = \{(q_0, 1, \rightarrow), (q_0, 0, \rightarrow)\}.$$

Construyendo la máquina para CNF-SAT. Para esta máquina vamos a asumir que las codificaciones de valuaciones $C(\tau)$ a un conjunto de n proposiciones es un string binario $a1, \ldots, a_n$, donde $\tau(p_i) = 0$ si y solo si el string tiene un 0 en la i-ésima posición, a_i.

Lo que hacemos es lo siguiente:

- Asumimos que tenemos como input la códificación de una fórmula $C(\varphi)$.
- Primero verificamos que el string tiene la forma $C(\varphi)$, y luego construimos el string $1^n BC(\varphi)$, donde n es la cantidad de proposiciones en φ.
- Ahora llamamos a la máquina M_b. Se abren 2^n posibles ejecuciones.
- Cuando M_b termina, la enlazamos con la máquina que resuelve CNF-EVAL, tomando como input el string computado por M_b y $C(\varphi)$. Nota que el string escrito por M_b es, en efecto, una valuación posible para las proposiciones de φ.

Sea M_S la máquina descrita anteriormente. Para mostrar que acepta a CNF-SAT, tenemos que mostrar que φ es satisfacible si y solo sí M_S acepta con input $C(\varphi)$. Notar que no nos importa qué es lo que pasa con las fórmulas que no son satisfacibles, esto es clave en no-determinismo.

Si φ es satisfacible, hay una valuación τ que lo satisface, y por tanto hay una ejecución de M_b que produce el string $C(\tau)$. Esa ejecución termina aceptando, por que al llamar a la máquina que resuelve CNF-EVAL esta va a aceptar. Por otro lado, si esta máquina acepta, es por que hay al menos una ejecución para la que CNF-EVAL aceptó. Por construcción, el string computado por M_b en esa rama de la ejecución va a satisfacer a φ.

Luego, la máquina funciona en tiempo polinomial: la construcción del string 1^n se puede hacer con un par de pasadas por $C(\varphi)$, M_b toma n pasos y CNF-EVAL toma tiempo $O(m^2)$, donde m es el tamaño de φ. Sumando todo, la máquina funciona en tiempo $O(m + n + m^2)$.

Con esto construimos una máquina no-determinista que funciona en tiempo polinomial cuando acepta a un string, y tal que acepta a un string si y solo si ese string es $C(\varphi)$, con φ satsifacible.

2. ¿CNF-SAT en PTIME?

Si bien aún no podemos mostrar que SAT o CNF-SAT están en PTIME, vamos a mostrar algo bastante fuerte: si CNF-SAT está en PTIME, entonces PTIME=NPTIME. Esto nos va a dar una luz: puede que SAT esté en PTIME, pero si lo está, es algo difícil de demostrar, pues equivale a mostrar que todas las palabras aceptadas por máquinas no-deterministas en tiempo polinomial pueden ser aceptadas por una máquina determinista en tiempo polinomial.

El camino para establecer el resultado de arriba es el siguiente.

- Definimos las nociones de hardness y completitud. La noción de hardness nos va a servir para capturar la intuición de que un problema es tan difícil como cualquier problema en ese clase. La noción de completitud nos sirve para establecer los problemas más difíciles de una clase.

- Vamos a mostrar después que si un problema es NP-completo y además está en PTIME, entonces PTIME=NP.
- Vamos a mostrar que CNF-SAT es NP-completo.

2.1. Hardness y completitud

El primer concepto es el de hardness, que da una noción de qué tan difícil es un problema con respecto a una clase. Intuitivamente, si un lenguaje L es hard para una clase de complejidad C, entonces L va a ser al menos tan difícil como cualquier lenguaje que pertenezca al conjunto C.

Definición. Sea C un conjunto de lenguajes que contiene a PTIME. Decimos que un lenguaje L es *hard* para C si para todo $L' \in C$ existe una reducción polinomial de L' a L.

El segundo concepto es el de completitud. Nos va a servir para saber la complejidad exacta de un problema.

Definición. L es completo para C si L es hard para C y a la vez $L \in C$.

Obviamente, si PTIME $\subsetneq C$ y L es completo para C entonces $L \notin$ PTIME.

Si L es hard para C, decimos que L es C-hard. Si es completo para C decimos que es C-completo.

Finalmente, podemos ver cómo nos sirven estas nociones, con la siguiente proposición.

Teorema. Sea C una clase de complejidad, tal que PTIME $\subseteq C$. Si un lenguaje L es C-completo y además L está en PTIME, entonces $C =$ PTIME.

Demostración. Para mostrar que $C =$ PTIME solo tenemos que probar que $C \subseteq$ PTIME (el otro lado lo tenemos del enunciado del Teorema). Sea entonces un lenguaje L_1 arbitrario que esté en C. Para mostrar que está en PTIME, tenemos que mostrar que hay una máquina de turing determinista, que funciona en tiempo polinomial, que acepta a L_1. La máquina se construye de la siguiente forma:

- Como L es C-hard, por definición hay una reducción polinomial f de L_1 a L. Supongamos que esa reducción funciona en tiempo n^k, es decir, hay una máquina M_f que la computa en ese tiempo.
- Como L está en PTIME, por definición hay una máquina de turing determinista M_L que acepta a L. Supongamos que esa máquina funciona en tiempo n^ℓ.
- La máquina para L_1 hace lo siguiente. Con input w, primero llama a M_f para computar $f(w)$. Luego le pasa $f(w)$ a M_L. Si M_L acepta a $f(w)$ entonces aceptamos a w, de lo contrario (si M_L no acepta a $f(w)$) no aceptamos a w.

Dado que (por definición de reducción) w pertenece a L_1 si y solo si $f(w)$ pertenece a L, tenemos que la máquina descrita arriba acepta a L_1. Más aún, lo hace en tiempo $((n^k)^\ell) = n^{k\ell}$, lo que es polinomial.

2.2. SAT es NP-completo, y 3-colorabilidad también!

El primer teorema no lo vamos a demostrar, pero es uno de los resultados más importantes del siglo pasado en computación. Ojo que ya sabemos cómo mostrar que SAT y CNF-SAT están en NP, lo que es mucho más difícil es mostrar que sonNP-hard!

Teorema (Cook-Levin). SAT, CNF-SAT y 3-CNF-SAT son NP-completos.

Ahora ya tenemos un lenguaje NP-completo, podemos pasar a demostrar otros lenguajes NP-completos. Para eso vamos a ayudarnos de un pequeño lema.

Lema. Sea C una clase de complejidad y L_1 y L_2 dos lenguajes. Si L_1 es C-hard, y además hay una reducción polinomial desde L_1 a L_2, entonces L_2 es C-hard también.

Demostración. Para mostrar que L_2 es C-hard, sea L_3 un lenguaje arbitrario en C. Tenemos que mostrar que hay una reducción polinomial desde L_3 a L_2.

Como L_1 es C-hard, hay una reducción, digamos f, desde L_3 a L_1. Además, sea h la reducción polinomial desde L_1 a L_2. Tenemos que componiendo h y f en $h \circ f$ obtenemos la reducción de L_3 a L_2.

Para demostrar que $h \circ f$ es una reducción polinomial, tenemos dos cosas que hacer:

- Mostrar que $h \circ f$ es polinomial. Esto sale directo de la composición de dos polinomios: Si f toma tiempo n^k y h toma tiempo n^ℓ, entonces $h \circ f$ toma tiempo $(n^k)^\ell = n^{k\ell}$.
- Mostrar que w pertenece a L_3 si y solo si $h(f(w))$ pertenece a L_2. Sabemos que w pertenece a L_2 si y solo si $f(w)$ pertenece a L_1. Además, para cualquier palabra v, v pertenece a L_1 si y solo si $h(v)$ pertenece a L_2. Haciendo $f(w) = v$, tenemos que $f(w)$ pertenece a L_1 si y solo si $h(f(w))$ pertenece a L_2, que era lo que buscábamos.

Como la elección de L_3 fue arbitraria, acabamos de mostrar que para cualquier lenguaje L en C hay una reducción desde L a L_2, lo que prueba que L_2 es hard.

Ahora si. Supongamos una codificación estándar $C(G)$ para grafos $G = (V, E)$ en strings binarios[1]. Definimos el lenguaje de los grafos 3-coloreables como

$$3\text{COL} = \{w \mid w = C(G) \text{ y } G \text{ es 3-coloreable}\}$$

Proposicion (Karp) . El lenguaje 3COL es NP-completo.

Demostración. Tenemos que mostrar que 3COL está en NP y además que 3COL es NP-hard. Para mostrar que 3COL está en NP, procedemos como en SAT:

- Construimos una máquina sobre el lenguaje $\{r, b, a, 0, 1\}$.
- Primero revisamos que w sea la códificación de un grafo $C(G)$, y escribimos $1^n 0 C(G)$, donde n es la cantidad de nodos de G.

[1]Una estándar: si G tiene n nodos $v_1, \ldots, v_n$, entonces $C(G) = 1^n 0 v$, donde v es un string de tamaño n^2 y tal que la posición i-ésima de v es un 1 cuando la arista (v_p, v_ℓ) está en E, donde p y ℓ son los enteros en $[1, n]$ tal que $i = n * p + \ell$.

- Luego *adivinamos la coloración*: invocamos una máquina de turing no-determinista que reemplaza 1^n con todos los posibles strings de largo n construibles con r, b o a. Esos strings son la codificación de una coloración.

- Enlazamos esa máquina con una máquina que chequee que esa coloración es válida (es decir que asigne colores distintos a los nodos conectados por una arista).

Para mostrar que $3COL$ es NP-hard, vamos a hacer una reducción de CNF-SAT a $3COL$. Esta reducción, con input $C(\varphi)$, computa la codificación $C(G)$ de un grafo G, de forma que φ es satisfacible si y solo si G es 3-coloreable. Esta reducción es parte de un video.

Listo! ya tenemos dos problemas NP-completos. No sabemos si hay algoritmos polinomiales para estos problemas, pero lo que si sabemos es que responder eso no es fácil: equivale a responder si acaso P = NP.

3. Ejercicios propuestos

El siguiente resultado es un poco más fuerte que lo que está en las notas, pero es igual de útil. La demostración es muy similar, por lo que es un buen ejercicio.

Teorema. Sean C y C' clases de complejidad tales que $\text{PTIME} \subseteq C \subseteq C'$. Si un lenguaje L es C'-completo y además L está en C, entonces $C = C'$.

Considera el lenguaje de 4-coloración: son todos los grafos en los que puedo pintar los nodos con uno de cuatro colores, y tal que ninguna arista tiene sus dos nodos con el mismo color. Muestra que el lenguaje 4COL es NP-completo:

$$\text{4COL} = \{w \mid w = C(G) \text{ y } G \text{ es 4-coloreable}\}$$