Práctico 3 Parte 1 Complejidad Computacional

Julián Viera

jviera@fing.edu.uy
julviera44@gmail.com

12 de octubre de 2021

Agenda

La clase de lenguajes **PSPACE**(Polynomial Space)

$$PSPACE = \bigcup_{c>0} SPACE(n^c)$$

Un lenguaje L pertenece a PSPACE si y solo si existe una MT M y dos constantes $c_1 > 0$ y $c_2 > 0$ tal que M decide L empleando a los sumo $c_1 n^{c_2}$ celdas de sus cintas de trabajo, a lo largo de toda su ejecución (se excluyen las celdas de la cinta de entrada).

Ejercicio 1 - Planteo del problema

El juego japonés $GO - MOKU$ generalizado es jugado por dos jugadores "X" y "O" en un tablero cuadrado $nxn, n > 5$. Los jugadores se turnan insertando sus marcas en el tablero, y el primer jugador que consiga tener 5 marcas consecutivas (adyacentes) en una fila, columna o diagonal gana el juego.
Una *posición* del tablero consiste en una distribución dada de marcas sobre el tablero, conjuntamente con la indicación de a que jugador le corresponde jugar (quién tiene el turno).

Sea $GM = \{< B > / B$ es una posición del juego $GO - MOKU$, en donde el jugador "X" tiene una **estrategia ganadora**$\}$
"X" tiene una estrategia ganadora para una posición B dada si hay una secuencia de movimientos (marcas) para "X" que le hacen ganar el juego, para cualquier secuencia de marcas que haga el jugador O.

El problema planteado consiste en probar que $GM \in PSPACE$.

Modelado de una **posición** del juego

Cada posición posible del juego lo modelamos como una pareja (B, T), en donde B es un vector de n^2 elementos que representa el estado actual del tablero (Board) y $T \in \{X, O\}$ indica a que jugador le corresponde el turno de jugar.
Cada elemento del vector $B(i) \in \{X, O, \square\}, i = 1..n^2$.

En una implementación a nivel de máquina de Turing, se podría representar el tablero como n^2 celdas consecutivas, con un 1 indicando la marca del jugador "X", un 0 la marca del jugador "0" y un blanco ($\square$) señalando una posición libre del tablero.

Un algoritmo para el problema

La idea es construir un programa **recursivo** $M(B, T)$ que devuelve 1 si el jugador "X" tiene una estrategia ganadora y 0 en caso contrario.
Si el turno T es para el jugador "X", se prueba sucesivamente a colocar la marca "X" en cada una de las posiciones libres del tablero B hasta encontrar una que corresponda a una estrategia ganadora o haber probado todas las posiciones libres.
Para cada posición que se marca con "X", se prueba a colocar sucesivamente la marca "O" del segundo jugador en **todas** las posiciones libres que quedan en el tablero, y para que la posición marcada con "X" pertenezca a una estrategia ganadora, la posición resultante luego de cualquier movida del jugador "O" debe ser también una posición con estrategia ganadora para "X", de ahí la formulación recursiva de la solución.

Un algoritmo para el problema

Si el turno T es para el jugador "O", se prueba sucesivamente a colocar la marca "O" en cada una de las posiciones libres del tablero B, verificando si para cada una de ellas existe al menos una estrategia ganadora para el jugador "X".
Si para alguna elección inicial marcada con "O" no hay en el tablero resultante una posición que se marque con "X" y que pertenezca a una estrategia ganadora para "X", el algoritmo devuelve 0.
Si para toda elección inicial marcada con "O" hay en el tablero resultante una posición que se marque con "X" y que pertenezca a una estrategia ganadora para "X", el algoritmo devuelve 1.

Pseudocodigo del algoritmo $M(B, T)$

Si tablero lleno ➔ ***reject*** (0)
Si T = X

① Para cada posición libre i en B $(B(i) = \square)$

$B \xrightarrow{B(i)=X} B'$

Si hay 5 X's alineados (H,V oD) en B' ➔ ***accept*** (1)
sino
Si tablero lleno ➔ ***reject***
sino

Para cada posición libre j en B' $(B'(j) = \square)$

$B' \xrightarrow{B'(j)=O} B''$

Si hay 5 O's alineados en B''
Procesar siguiente i (ir a ①)
sino
Si $M(B'', X)$ = *reject*
Procesar siguiente i (ir a ①)
Fin Para cada j libre
accept (existe i tal que $B(i) = X$ es parte de estrategia ganadora)

Fin Para cada i libre
reject (no existe i tal que $B(i) = X$ es parte de estrategia ganadora)

Pseudocodigo del algoritmo $M(B, T)$

sino (T = O)

② Para cada posición libre i en B $(B(i) = \square)$

$B \xrightarrow{B(i)=O} B'$

Si hay 5 O's alineados en B' ➔ ***reject***

sino

si tablero lleno ➔***reject***

sino

Para cada posición libre j en B' $(B'(j) = \square)$

$B' \xrightarrow{B'(j)=X} B''$

Si hay 5 X's alineados en B''

Procesar siguiente i (ir a ②)

sino si $M(B'', O)$ = *accept*

Procesar siguiente i (ir a ②)

Fin Para cada j libre

reject (X no tiene estrategia ganadora)

Fin Para cada i libre

accept ($B(j) = X$ es parte de estrategia ganadora para X)

Espacio utilizado por el algoritmo $M(B, T)$

El espacio máximo utilizado por el algoritmo viene dado por la máxima cantidad de llamadas recursivas **anidadas** para las cuales hay que guardar información en el stack.
Como hay n^2 posiciones posibles, y se marcan 2 posiciones del tablero entre llamadas consecutivas, hay en total en el peor caso $\mathcal{O}(\frac{n^2}{2})$ llamadas anidadas.
Por cada llamada se deben guardar el estado de las n^2 posiciones del tablero, así como los índices i y j, es decir se necesita un espacio $\mathcal{O}(n^2)$ por llamada recursiva.
$\Longrightarrow$ el algoritmo propuesto es $\mathcal{O}(n^4)$ en espacio.
$\Longrightarrow$ $GM \in PSPACE$.

El problema *PATH*

$PATH = \{< G, s, t > / G$ es un grafo dirigido en el cual hay camino entre los nodos s y $t\}$

Puede demostrarse que $PATH \in NSPACE(log(n))$ ($PATH \in NL$).

El teorema de Savitch

Para cualquier función espacio-construible (space-constructible) $S : N \to N$ con $S(n) \geq \log n$, $NSPACE(S(n)) \subseteq SPACE(S(n)^2)$

Planteo del problema

Se define la clase **PolyL**$= \bigcup_{c>0} SPACE(\log^c n)$.

La clase SC ("Steve Class",nombrada en honor a Steve Cook) se define como el conjunto de lenguajes que pueden ser decididos por MT deterministas que corren en tiempo polinomial y en espacio $\log^c n$ para algún $c > 0$.

Es un problema abierto si el lenguaje $PATH \in SC$.

1. ¿Por qué el teorema de Savitch no resuelve esta cuestión?
2. ¿Es SC lo mismo que $PolyL \bigcap P$?

¿Por qué el teorema de Savitch no resuelve si $PATH \in SC$?

Sabemos que $PATH \in NSPACE(\log n)$ (la demostración está en Arora-Barak) $\xrightarrow{\text{teorema de Savitch}} PATH \in SPACE(\log^2 n)$.

Es decir que sabemos que existe una MT determinista M que decide $PATH$ en espacio $\log^2 n$ y por lo tanto $PATH \in PolyL$.

La razón por la que el teorema de Savitch no asegura que $PATH \in SC$ es que dicho teorema no dice nada acerca del tiempo de ejecución de M, es decir que no garantiza que dicho tiempo sea polinomial, tal como lo requiere la definición de SC.

¿Es SC lo mismo que $PolyL \bigcap P$?

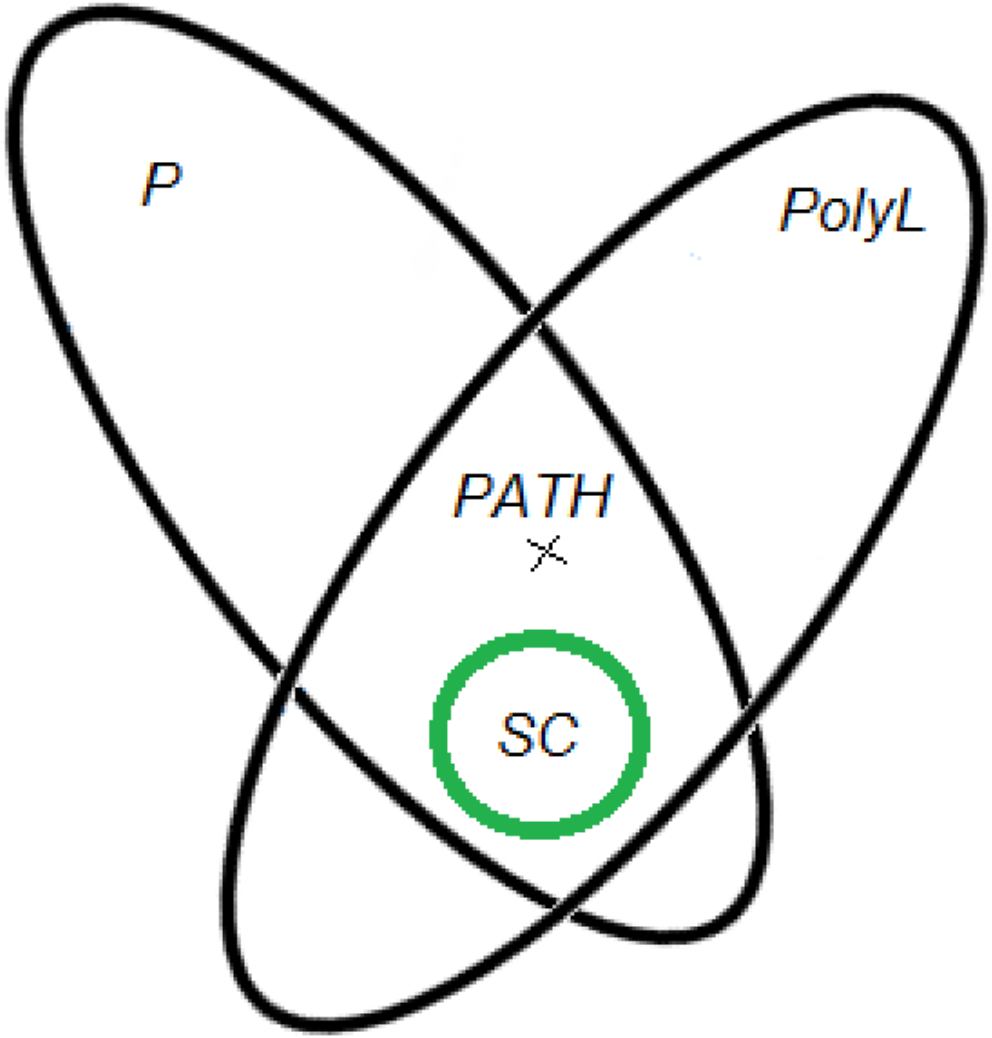
La respuesta a esta cuestión es **no necesariamente**.

Si $L \in SC \Longrightarrow L \in P$ y $L \in PolyL \Longrightarrow SC \subseteq PolyL \bigcap P$.

Sea $L \in PolyL \bigcap P$. Puede darse el caso de que la MT A que decide L en espacio $\log^c n$ no ejecute en tiempo polinomial, y que la MT B que decide L en tiempo polinomial no se ejecute en espacio $\log^c n$. En este caso no podemos asegurar que $L \in SC$, y por lo tanto que $PolyL \bigcap P \subseteq SC$.

Consideremos por ejemplo el caso de $PATH$. Sabemos que $PATH \in P$, pero los algoritmos polinómicos en tiempo para $PATH$ no son de espacio $\log^c n$. Por ejemplo un algoritmo polinómico de búsqueda en profundidad (Depth First Search) requiere espacio $\mathcal{O}(n)$, ya que hay que mantener una lista de los vértices ya visitados.

Una posible relación entre las distintas clases del problema



Problemas *PSPACE – hard*

Un lenguaje L^* es PSPACE-hard si para todo lenguaje $L \in PSPACE$, se verifica $L \leq_P L^*$.

Si además se cumple que $L^* \in PSPACE$, se dice que L^* es PSPACE-completo.

Planteo del problema

Mostrar que si todo problema NP-hard es PSPACE-hard, entonces PSPACE = NP

Idea de la demostración: por doble inclusión, probando que

1. $NP \subseteq PSPACE$
2. $PSPACE \subseteq NP$

Prueba de que $NP \subseteq PSPACE$

① Sabemos que $NP \subseteq NPSPACE$
② Por el teorema de Savitch, $NPSPACE = PSPACE$

De ① y ② $\Longrightarrow NP \subseteq PSPACE$

Esta inclusión se cumple siempre, y es independiente de la hipótesis de este problema.

Prueba de que $PSPACE \subseteq NP$

Sea L un problema NP-completo cualquiera.

Como L es NP-hard $\overset{\text{H}}{\Longrightarrow}$ L es PSPACE-hard.

Sea $L^* \in PSPACE$.
Como L es PSPACE-hard$\Longrightarrow$ $L^* \leq_P L$, es decir que $\exists f$ construible en tiempo polinómico tal que $x \in L^* \Leftrightarrow f(x) \in L$.

Como f es computable en tiempo polinómico, se verifica también que $|f(x)| \leq q(|x|)$, siendo q el polinomio asociado a la transformación f. Tenemos entonces que el tamaño (espacio ocupado) por $f(x)$ está acotado polinomialmente en el tamaño de la instancia x.

Como $L \in NP \Longrightarrow$ existe un polinomio p y una MT M verificadora con polinomio asociado s de forma que:
$\forall \omega \in \{0,1\}^*, \omega \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|\omega|)} / M(\omega, u) = 1$

Prueba de que $PSPACE \subseteq NP$

Para probar que $L^* \in NP$ se dará un certificado y un sistema de prueba para cada instancia $x \in L^*$.
Dado $x \in L^*$, se toma como certificado u para x al certificado asociado a $f(x)$ en L.
Observamos que $|u| = \{0,1\}^{p(q(|x|))}$ es polinomial en $|x|$.
La MT verificadora $M^*(x,u)$ para L^* se define como sigue:
Calcular $f(x)$
Si $M(f(x),u) = 1 \Longrightarrow M^*(x,u) = 1$
sino $M^*(x,u) = 0$
Como $x \in L^* \Leftrightarrow f(x) \in L$, se verifica que $x \in L^* \Leftrightarrow M^*(x,u) = 1$
M^* se ejecuta en tiempo polinomial en $|x|$ ya que f lo hace y M también. El tiempo de ejecución de M vale
$s(|f(x)| + |u|) = s(|q(|x|)| + p(q(|x|)))$, polinomial en $|x|$.
$\xrightarrow{\text{definición de NP}} L^* \in NP$
Como se probó la doble inclusión $\Longrightarrow PSPACE = NP$.

La clase de lenguajes **L** (Logarithmic Space)

$L = SPACE(\log n)$

Es la clase de lenguajes que pueden ser decididos por una MT M determinista que usa **espacio logarítmico** en sus cintas de trabajo.

Planteo del problema

Sea el lenguaje: $PAL - ADD = \{< x, y > / x, y > 0$ enteros binarios y $\llcorner x + y \lrcorner$ es un palíndromo$\}$

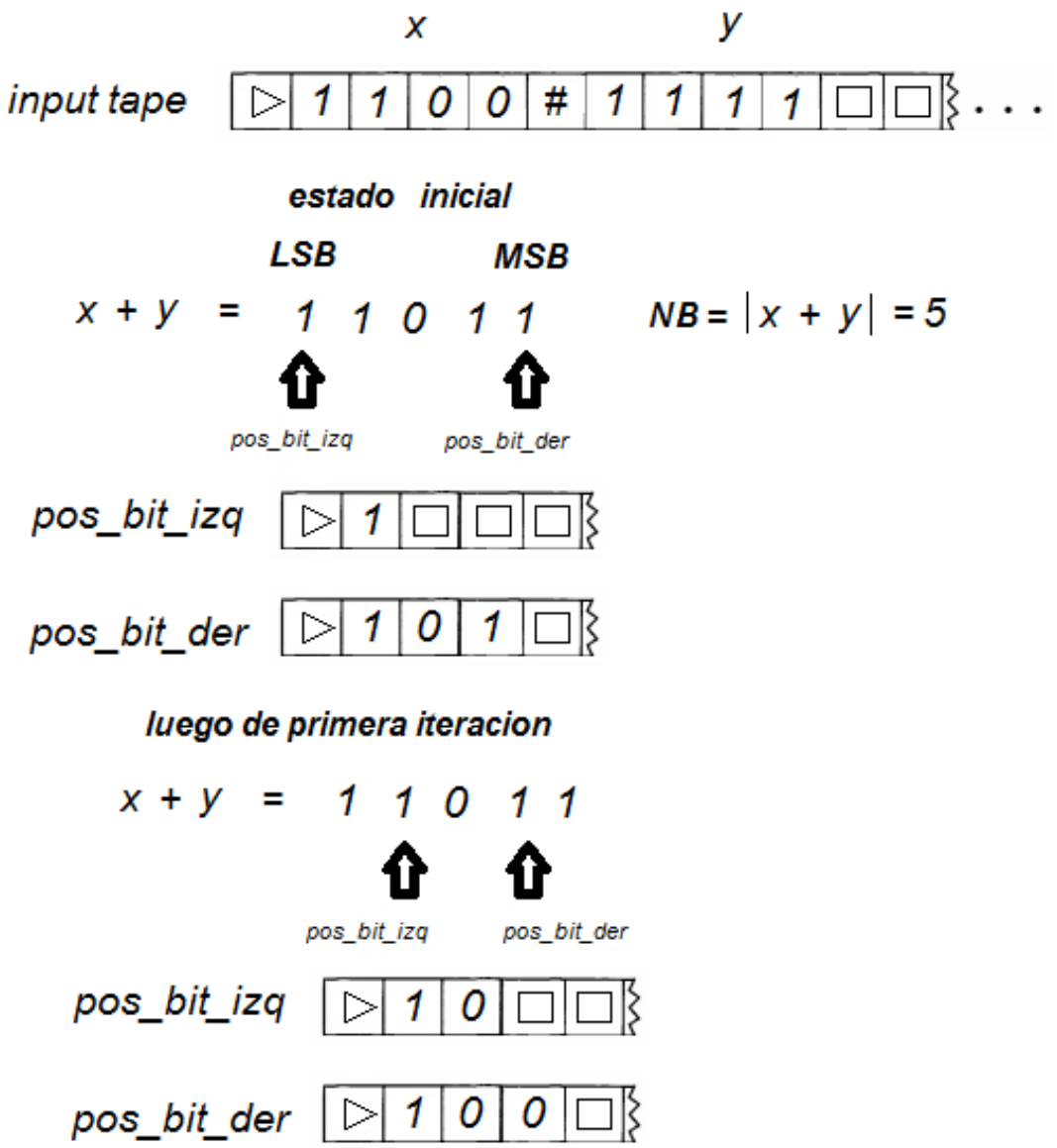
Es el lenguaje de las parejas de enteros positivos tales que la representación binaria de su suma es un palíndromo. Se asume que la representación binaria de la suma no tiene ceros no significativos

Probar que $PAL - ADD \in L$.

Un algoritmo para el problema

Trabajamos con la representación "invertida" de x+y, que tiene el bit menos significativo a la izquierda. La idea es mantener dos contadores con las posiciones de bits en posiciones simétricas respectos a los extremos del string suma x+y. Se tiene un contador *POS_BIT_IZQ* que marca la posición actual del bit de la izquierda, y un contador *POS_BIT_DER* que marca el bit de la derecha correspondiente en x+y. En cada paso del algoritmo se calcula el bit de la suma en la posición *POS_BIT_IZQ* y el bit de la suma en la posición *POS_BIT_DER* y se los compara. Si son distintos, el algoritmo devuelve 0 pues la suma no es palindromo. Si son iguales, se incrementa en uno el contador *POS_BIT_IZQ* y se decrementa en uno el contador *POS_BIT_DER*. Si los contadores son iguales o se cruzan, significa que la suma es palindromo y el algoritmo devuelve 1. En caso contrario se repite el procedimiento. El valor de *POS_BIT_IZQ* se inicializa en 1 y el de *POS_BIT_DER* con el número de bits de la suma x+y, que se calcula al principio.

Ejemplo de funcionamiento del algoritmo



Pseudocódigo del algoritmo

Calcular el numero de bits NB en la representacion binaria de x+y
pos_bit_izq = 1
pos_bit_der = NB
Repetir
    Calcular *bit-izq* , el bit de la suma en la posición *pos_bit_izq*
    Calcular *bit-der* , el bit de la suma en la posición *pos_bit_der*
    Si *bit-izq* ≠ *bit-der* PARAR y devolver 0
    sino
        incrementar *pos_bit_izq*
        decrementar *pos_bit_der*
        Si *pos_bit_izq* ≥ *pos_bit_der*
            PARAR y devolver 1
        Fin Si
    Fin Si
Fin Repetir

Espacio utilizado por el algoritmo

Los contadores POS_BIT_IZQ y POS_BIT_DER ocupan espacio logarítmico en $n = |x| + |y|$ (en el ejemplo, $|x| + |y| = 8$ y los contadores requieren 3 bits).

El contador que se usa para calcular el bit de la suma que está en la posición i-ésima también ocupa $\mathcal{O}(log(|x| + |y|))$.

$\Longrightarrow PAL - ADD \in L$.