Apunte de Complejidad Computacional

Santiago Figueira

Departamento de Computación, FCEN, UBA
19 de junio de 2025

Resumen

Este es el apunte de la materia *Complejidad Computacional*, de la Licenciatura en Ciencias de la Computación (FCEN, UBA). Va de la mano con las slides que vemos en las clases teóricas. Verán que este documento tiene gran similitud con ellas, y que aparecen muchos dibujitos y esquemas, quizá a veces un poco repetitivos. Traté de escribirlo en un tono bien didáctico, pero también bien formal. La ventaja respecto a las slides es que acá explico en más detalle algunas cosas que prefiero no explicar en la clase. Además, acá tienen todas las referencias a las definiciones y resultados que se van citando a lo largo del desarrollo, y eso es algo que no aparece en las slides.

Casi todo el material está tomado de [1]. En la mayoría de los casos cambié las demostraciones porque me pareció que faltaba profundidad y formalidad. Para algunos temas me guié con [4] y [5]. Otro libro de consulta fue [3].

Esta es la primera versión de este apunte. Espero que no tenga muchos errores serios, pero seguro que tiene varias cuestiones menores. Si ven erratas, partes que no se entienden, contradicciones entre definiciones (van a ver que hay muchas definiciones), etc. me pueden ayudar a ir mejorándolo, escribiéndome a `santiago@dc.uba.ar`.

☞ Este pdf contiene links internos: si cliquean en una noción, los lleva a una **definición**.[1]

Índice

[1] `https://ctan.org/pkg/knowledge`

1. Introducción

En la materia *Lenguajes Formales, autómatas y computabilidad* ustedes ya estudiaron varios modelos de cómputo. En esta materia nos centramos en el más fuerte de todos: las máquinas de Turing, y sus variantes, aunque hacia le final estudiaremos también el modelo de circuitos booleanos. Mientas la Teoría de la Computabilidad se centra en el estudio de los grados de dificultad de las funciones no computables, la Teoría de la Complejidad Computacional se centra exclusivamente en el estudio de los grados de dificultad de funciones computables. En el modelo de las máquinas de Turing —pero también en cualquier lenguaje de programación actual—, la dificultad se puede medir de dos formas muy naturales para un programador: la cantidad de tiempo o la cantidad de espacio que toma una implementación. El tiempo se refiere a la cantidad de pasos que toma realizar un cómputo. El espacio se refiere a la cantidad de máxima de celdas de memoria que se usa a lo largo del cómputo.

La Teoría de la Complejidad Computacional estudia varias clases de problemas importantes para la computación. Muchas veces se conoce la inclusión entre una clase y otra, pero se desconoce si las dos clases son distintas o si en realidad son la misma. Por ejemplo, cuando hablamos del tiempo, se desconocen muchas cotas inferiores al tiempo que tiene que tomar la mejor solución posible a un cierto problema. Es importante aclarar dos cosas. Una es que hablamos de problemas

y no de algoritmos concretos; calcular el tiempo exacto que toma un algoritmo particular en llegar a la solución puede ser difícil, pero no es nuestro objeto de estudio. La otra es que no deben tomar eso de la 'mejor solución posible' al pie de la letra; las cotas de tiempo y espacio que vamos a estudiar son aproximadas, grosso modo, algo que se formaliza con la notación de O grande y de o chica. En general alcanza con encontrar soluciones, es decir, algoritmos, 'razonablemente buenos' para probar que un problema está en una cierta clase.

Esta falta de conocimiento respecto de las inclusiones estrictas o no de las clases contrasta con el escenario en la Teoría de la Computabilidad, donde se conocen separaciones de muchas clases. La idea de diagonalización, por ejemplo, es una herramienta poderosa que permite separar muchas clases en Computabilidad. Sirve para construir objetos de manera computable. Pero cuando la eficiencia entra en juego, muchos argumentos de la Teoría de la Computabilidad, dejan de valer (otros, por suerte, como algunos usos de la diagonalización, siguen valiendo).

En la materia *Técnicas de diseños de algoritmos* estudiaron clases de complejidad importantes, como **P** y **NP**. En esta materia formalizaremos estas definiciones, estudiaremos más en detalle algunos conceptos vistos en esa materia y, lo más importante, veremos nuevas clases de complejidad. Esta es una materia teórica. Los teoremas que veremos hablarán de inclusión de clases, de igualdad de clases y de separación de clases, de reducciones.

Presupongo cierta madurez en el pensamiento algorítmico. Abundarán las descripciones informales de algoritmos, y mostraremos pseudocódigos sin extrema rigurosidad. Siempre quedará claro, eso sí, qué computan, qué construyen, cómo funcionan. La Teoría de la Complejidad Computacional no estudia los algoritmos en sí, ni la aplicación de las estructuras de datos, sino la relación entre clases de problemas (que se formalizan como lenguajes, como los que vieron en *Lenguajes Formales, autómatas y computabilidad*). Definir completamente una máquina de Turing para probar que un cierto problema se resuelve con el tiempo acotado que buscamos, o con el espacio acotado que buscamos, haría el curso muy largo, aburrido y, sobre todo, desviaría la atención de los resultados. Podría parecer una contradicción el hecho de empezar definiendo con tanto detalle un modelo de cómputo, el de las máquinas de Turing, para después usarlo solo informalmente. Puede ser que lo sea, pero creo que es la única forma de meter en un curso los contenidos básicos de la Teoría de la Complejidad Computacional.[2] No buscamos entrenar programadores de máquinas de Turing sino explicar la estructura de las clases de dificultad de los problemas. Así, daremos descripciones informales y de alto nivel de los programas y, cuando haga falta, aclararemos el uso de algún truco para la implementación real, y, lo más importante, siempre daremos argumentos de que la implementación real preserva los requerimientos acerca del uso de los recursos de tiempo o espacio que necesitamos probar. Entonces, cada vez que lean un pseudocódigo en *verde*, deben convencerse de que se puede transformar (con suficiente dedicación y paciencia) a una máquina de Turing con las características buscadas.

1.1. Notación

Un **alfabeto** es un conjunto de símbolos. Salvo que se diga lo contrario, un alfabeto siempre será finito. Dado un alfabeto Γ, notamos Γ^* al conjunto con todas las palabras finitas formadas sobre Γ. Por ejemplo, si $\Gamma = \{0,1\}$, entonces $\Gamma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$. El símbolo ϵ representa la palabra vacía. Si $\sigma \in \Gamma^*$ entonces $|\sigma|$ es la longitud de σ, es decir la cantidad de símbolos de σ. Por ejemplo, si $\Gamma = \{0,1\}$, $\sigma = 101$, y $\tau = \epsilon$ entonces $|\sigma| = 3$, y $|\tau| = 0$. Si $n \leq |\sigma|$, definimos $\sigma \upharpoonright n$ como los primeros n símbolos del σ. Por ejemplo, si $\sigma = 111001$ entonces $\sigma \upharpoonright 4 = 1110$. Notar que $\sigma \upharpoonright 0 = \epsilon$ para toda σ. También usaremos $|X|$ para referirnos a la cantidad de elementos del conjunto X. Siempre quedará claro del contexto a qué definición de $|\cdot|$ nos referimos. Γ^k es el conjunto con todas las palabras formadas sobre Γ de longitud k. Por ejemplo, si $\Gamma = \{0,1\}$, entonces $\Gamma^2 = \{00, 01, 10, 11\}$. También podemos ver a Γ^2 como conjunto de tuplas, $\Gamma^2 = \{(0,0), (0,1), (1,0), (1,1)\}$. Siempre va a quedar claro del contexto cuál definición usamos, aunque en realidad las dos son equivalentes. Definimos $\Gamma^{\leq k} = \bigcup_i \leq k \Gamma^i$. Si $\sigma, \tau \in \Gamma^*$,

[2] Ningún libro de Complejidad Computacional o Computabilidad detalla las construcciones hasta el punto de definir las máquinas, sino que se limitan a dar pistas, a dar argumentos convincentes sobre la existencia de tales máquinas.

entonces $\sigma\tau$ representa la concatenación de σ con τ. Si $\sigma \in \Gamma^*$, $i \in \{0, \ldots, |\sigma| - 1\}$, entonces $\sigma(i)$ es la $(i+1)$-ésima letra de σ. Por ejemplo, si $\Gamma = \{0, 1\}$, y $\sigma = 101$ entonces $\sigma(0) = 1$, $\sigma(1) = 0$, $\sigma(2) = 1$.

1.2. Codificación en binario

En general trabajaremos con el alfabeto binario, y con máquinas que procesan palabras en binario. La elección de que sea binario es arbitraria. Lo hacemos solo para fijar un alfabeto, pero lo mismo podríamos hacer con cualquier alfabeto finito. De hecho, veremos máquinas de Turing que trabajan con alfabetos arbitrarios. En general, los resultados que veremos son independientes de la elección del alfabeto (salvo para el caso del alfabeto unario).

Vamos a trabajar con varios objetos matemáticos: cadenas, números, grafos, y, en general, estructuras que se puedan representar finitamente usando un alfabeto finito. Estos objetos van a ser entradas o salidas de métodos efectivos, que formalizaremos más adelante con las máquinas de Turing. Entonces, para uniformizar todos los resultados, vamos a usar el alfabeto binario $\{0, 1\}$ para representar todos estos objetos.

Codificación de números. Si $n \in \mathbb{N}$, notamos $\boldsymbol{[n]}$ a la representación de n en binario y notamos $\boldsymbol{|n|}$ al tamaño de $[n]$, es decir $|n| = \lceil \log_2 n \rceil$. Escribiremos $\log n$ en vez de $\log_2 n$. A veces vamos a usar $\log n$ como abreviatura de $\lceil \log n \rceil$. Por ejemplo $[0] = 0$ ($|0| = 1$) y $[5] = 101$ ($|5| = 3$). En algunas ocasiones vamos a representar a los números en **unario**. Esto quiere decir que representamos $n \in \mathbb{N}_{>0}$ como una secuencia de n unos. Por ejemplo, la codificación del 1 en unario es 1 y la codificación de 5 en unario es 11111. Siempre que elijamos esta representación, vamos a avisarlo explícitamente. Si $b \in \Gamma$ y $n \geq 0$, notamos b^n a $b \ldots b$ (n veces). Por ejemplo. $1^5 = 11111$.

Codificación autodelimitante de cadenas. Supongamos que $\sigma, \tau \in \{0, 1\}^*$. No podemos codificar el par (σ, τ) como $\sigma\tau$ porque no queda claro dónde termina σ y dónde empieza τ. Vamos a usar codificaciones autodelimitantes, que codifican no solo la cadena sino también donde termina la cadena. Para $\sigma = b_1 b_2 \ldots b_n l$ ($b_i \in \{0, 1\}$), definimos

$$\boldsymbol{\langle \sigma \rangle} = b_1 b_1 \; b_2 b_2 \; \ldots \; b_n b_n \; 10$$

(el 10 marca el final). Por ejemplo, $\langle 101 \rangle = 11\ 00\ 11\ 10$ y $\langle \epsilon \rangle = 10$ (los espacios no forman parte de la codificación). Ahora podemos codificar el par (σ, τ) como $\langle \sigma \rangle \langle \tau \rangle$. Lo mismo se puede hacer con tuplas de cualquier dimensión.

Codificación de listas o conjuntos. Codificamos la lista $L = \sigma_1, \ldots, \sigma_n$ ($\sigma_i \in \{0, 1\}^*$) como

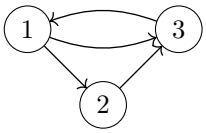
$$\langle L \rangle = \langle \sigma_1 \rangle \; \langle \sigma_2 \rangle \; \ldots \; \langle \sigma_n \rangle \; 01$$

(el 01 marca el fin de lista, el 10 dentro de cada $\langle \sigma_i \rangle$ marca el final de σ_i, y los espacios no forman parte de la codificación). Por ejemplo. $\langle 11, 100, 0001 \rangle = \langle (11, 100, 0001) \rangle = \langle \{11, 100, 0001\} \rangle = 111110\ 11000010\ 0000001110\ 01$ y $\langle \emptyset \rangle = 01$

Para simplificar la notación, muchas veces vamos a codificar la tupla de objetos $(O_1, \ldots, O_n)$ usando la notación $\langle O_1, \ldots, O_n \rangle$ en vez de $\langle \langle O_1 \rangle, \ldots, \langle O_n \rangle \rangle$. Del mismo modo, usaremos $|O|$ en vez de $|\langle O \rangle|$ para referirnos al tamaño de la codificación de O.

Codificación de grafos. Si $G = (E, V)$ es un digrafo, podemos representar G con la matriz de adyacencia

$$A = \begin{pmatrix} b_{11} & b_{12} & \ldots & b_{1k} \\ b_{21} & b_{22} & \ldots & b_{2k} \\ & & \ldots & \\ b_{\ell 1} & b_{\ell 2} & \ldots & b_{\ell k} \end{pmatrix} \qquad (b_{ij} \in \{0, 1\}^*)$$



$$\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad \begin{aligned} \langle G \rangle &= 00\ 11\ 11\ 10\ 00\ 00\ 11\ 10\ 11\ 00\ 00\ 10\ 01 \\ |\langle G \rangle| &= 26 \end{aligned}$$

Figura 1: Un digrafo, su matriz de adyacencia y su codificación (los espacios en $\langle G \rangle$ no forman parte de la codificación).

con

$$\begin{aligned} \langle A \rangle &= \langle \langle b_{11}, \ldots, b_{1k} \rangle \langle b_{21}, \ldots, b_{2k} \rangle \ldots \langle b_{\ell 1}, \ldots, b_{\ell k} \rangle \rangle \\ |\langle A \rangle| &= 2 \cdot \ell \cdot (k+1) + 2 \end{aligned}$$

(ver Figura 1).

1.3. Problemas de decisión

Los problemas se formalizan con funciones. Por ejemplo, el problema de sumar dos números x e y se formaliza con la función $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definida por $g(x, y) = x + y$. Podemos también pensarla como una función $h : \{0,1\}^* \to \{0,1\}^*$ definida por $h(\langle x, y \rangle) = \langle x+y \rangle$. Un caso particular de problemas son los **problemas de decisión**, que se formalizan como funciones booleanas, es decir funciones de la forma

$$f : \{0,1\}^* \to \{0,1\}$$

Se llaman problemas de decisión porque, por cada entrada, solo deciden por 'sí' (codificado con 1) o por 'no' (codificado por 0). Para abreviar, en general los llamaremos sencillamente problemas.

Un **lenguaje** es un conjunto de palabras sobre un alfabeto dado. Nosotros vamos a trabajar con lenguajes sobre el alfabeto $\{0,1\}$, aunque cambiar el alfabeto no implica ninguna dificultad adicional.

Cada función booleana f representa el lenguaje

$$\mathcal{L}(f) = \{x \colon f(x) = 1\} \subseteq \{0,1\}^*$$

Y viceversa: cualquier lenguaje $\mathcal{L} \subseteq \{0,1\}^*$ se puede representar por la función booleana

$$\chi_{\mathcal{L}}(x) = \begin{cases} 1 & \text{si } x \in \mathcal{L} \\ 0 & \text{si no} \end{cases} \tag{1}$$

Así, lenguajes o problemas de decisión son dos términos representan el mismo objeto.

2. Máquinas y cómputos

Vamos a fijar un modelo de cómputo, que puede interpretarse como un lenguaje de programación. Aunque parezca algo primitivo, nos va servir para definir formalmente qué es un cómputo y para contar la cantidad de operaciones que toma llevar adelante ese cómputo. Casi toda la teoría que desarrollaremos se basará en este modelo o en pequeñas variaciones de este modelo (los resultados de la teoría de la complejidad son bastante generales y usualmente independientes de las variaciones razonables del modelo de cómputo elegido para contar la cantidad de operaciones). Solo en §8 usaremos un modelo distinto, el de los circuitos booleanos, pero incluso los circuitos estarán muy vinculados al modelo que presentamos en esta sección.

El modelo que vamos a presentar se llama 'máquinas de Turing'. Es el modelo que propuso Alan Turing en su trabajo fundacional de 1936 para definir formalmente la idea de método efectivo, de procedimiento mecánico. Hoy sería más fácil pensar en formalizar esta idea: podríamos sencillamente decir que los métodos efectivos son los programables en algún lenguaje de programación que elijamos, por ejemplo en Python. Pero en 1936 no existía Python ni los lenguajes

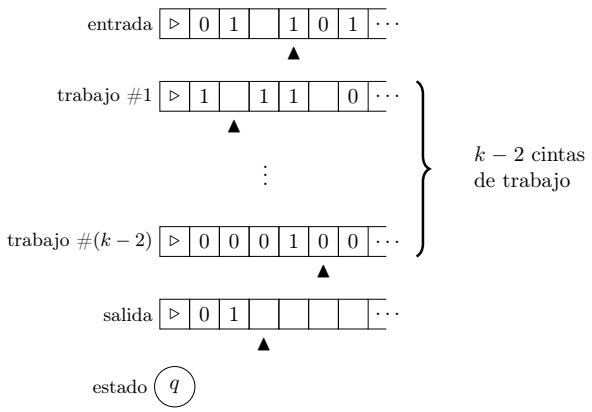


Figura 2: Una máquina con k cintas con celdas en estado q. Cada cinta tiene una cabeza (representada con el símbolo ▲) ubicada en alguna de las celdas. El alfabeto es el estándar.

.

de programación como los conocemos ahora. Entonces, el primer paso para estudiar los métodos efectivos era, precisamente, definirlos formalmente.

Una máquina de Turing es un dispositivo teórico, un modelo abstracto de cómputo. Se compone de:

- **Alfabeto.** Es un conjunto finito de símbolos. En general lo vamos a notar con Σ o con Γ. El alfabeto es arbitrario, pero siempre contiene dos símbolos especiales: $\triangleright$, que va a indicar el comienzo de la cinta y $\square$, que va a representar el blanco. Cuando $\Sigma = \{0, 1, \triangleright, \square\}$, hablamos del **alfabeto estándar**.

- **Cintas.** Tiene $k \geq 3$ cintas. Una cinta se llama de **entrada**, otra se llama de **salida** y las $k - 2$ restantes se llaman de **trabajo**. Cada cinta es infinita a derecha y está dividida en celdas. Cada celda contiene un símbolo del alfabeto Σ.

- **Cabezas.** Cada cinta tiene una cabeza que la recorre. Cada cabeza se puede mover hacia la derecha o hacia la izquierda, siempre de a una celda a la vez. La cabeza de la cinta de entrada puede leer pero no escribir; las cabezas de las cintas de trabajo pueden leer y escribir y la cabeza de la cinta de salida solo puede escribir. Lo que leen y escriben son siempre símbolos del alfabeto.

- **Estados.** Cada máquina tiene un conjunto finito de estados. En general lo vamos a notar como Q. En cada momento, la máquina estará en un estado de Q. El conjunto de estados Q siempre tiene dos estados distinguidos: q_0 y q_f. A q_0 se lo llama **estado inicial** y a q_f se lo llama **estado final**. La Figura 2 muestra una máquina con k cintas en un determinado momento.

- **Instrucciones.** Las instrucciones hacen evolucionar la máquina. Permiten que la máquina cambie de estado, mueva las cabezas hacia la izquierda o derecha y cambien el contenido de algunas celdas. Si Σ es el alfabeto y Q es el conjunto de estados, las instrucciones son siempre de esta forma:

  Si

  - *el estado es $q \in Q$,*
  - *la cabeza de la cinta de entrada está leyendo $a \in \Sigma$, y*
  - *la cabeza de la j-ésima cinta de trabajo está leyendo $b_j \in \Sigma$ ($j \in \{1, \ldots, k-2\}$),*

  entonces

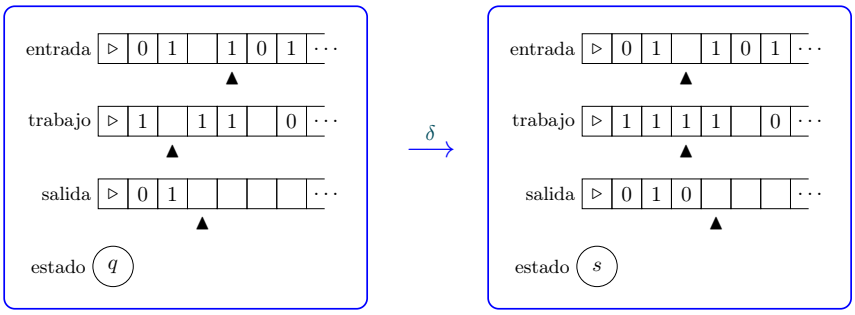


Figura 3: Una máquina que cambia su estado y sus cintas según $\delta(q, 1, \square) = (L, 1, R, 0, s)$.

.

- *en cinta de entrada, mover la cabeza a la izquierda* (L) *una posición, a la derecha* (R) *una posición, o bien dejarla en el mismo lugar* (S)*; codifiquemos esta acción con* $E \in \{R, S, L\}$
- *en cinta* j*-ésima cinta de trabajo* $(j \in \{1, \ldots, k-2\})$*: escribir* $t_j \in \Sigma$ *y mover la cabeza a la izquierda* (L) *una posición, a la derecha* (R) *una posición, o bien dejarla en el mismo lugar* (S)*; para cada* j*, codifiquemos esta acción con* $T_j \in \{R, S, L\}$
- *en cinta de salida: o bien escribir* $c \in \Sigma$ *y mover la cabeza hacia la derecha una posición, o bien no hacer nada; codifiquemos esta acción con* $Z = c$ *en el primer caso o* $Z = S$ *en el segundo*
- *pasar al estado* $q' \in Q$.

Cada instrucción de este tipo se puede codificar con una función

$$\delta : Q \times \Sigma^{k-1} \to \underbrace{\{L, R, S\}}_{\text{entrada}} \times \underbrace{\Sigma^{k-2} \times \{L, R, S\}^{k-2}}_{k-2 \text{ trabajos}} \times \underbrace{(\Sigma \cup \{S\})}_{\text{salida}} \times Q \tag{2}$$

que mapea $(q, a, b_1, \ldots, b_{k-2})$ a $(E, t_1, \ldots, t_{k-2}, T_1, \ldots, T_{k-2}, Z, q')$. La función δ se llama **función de transición** y tiene la información de cómo evoluciona la máquina paso a paso. Desde luego, la evolución dependerá del contenido de las cintas, pero queda claro que, fijados los contenidos de cada cinta y el estado, la máquina evoluciona de una única forma, y esa forma viene dada por δ. Observar que, dado que Σ y Q son conjuntos finitos, el dominio de δ también es finito. Así, δ es finitamente representable y puede pensarse como una tabla con $|Q| \cdot |\Sigma|^{k-1}$ filas y representa el programa de la máquina.

La Figura 3 muestra una máquina con 3 cintas antes y después de ejecutar la función de transición δ. La cabeza de la cinta de entrada está leyendo un 1, la cabeza de la (única) cinta de trabajo está leyendo un $\square$ y el estado es q. La función δ actúa y mueve la cabeza de entrada hacia la derecha, escribe un 1 y se mueve a la derecha en la cinta de trabajo y escribe un 0 y se mueve a la derecha en la cinta de salida.

Un comentario menor es que la función de transición tiene algunas restricciones para evitar que las cabezas intenten moverse a la izquierda del comienzo de la cinta de entrada[3]. Para una máquina de 3 cintas, esto quiere decir que $\delta(*, \triangleright, *)$ no puede ser de la forma $(L, *, *, *, *)$ y $\delta(*, *, \triangleright)$ no puede ser de la forma $(*, *, L, *, *)$. Esto se generaliza fácilmente para máquinas con k cintas.

Por otro lado, una vez que la máquina entra al estado q_f, se queda congelada. Es decir, a partir de ese momento δ no cambia nada –permanece en el mismo estado, no mueve las cabezas y no cambia el contenido de la celda apuntada por cada cabeza. Para una máquina de 3 cintas, esto quiere decir que $\delta(q_f, a, b)$ solo puede ser de la forma (S, b, S, S, q_f), y se generaliza fácilmente para

[3]En principio, el símbolo $\triangleright$ podría ser escrito en cualquier celda de las cintas de trabajo y salida, aunque esto es irrelevante. Lo que sí importa es que δ nunca borre los símbolos $\triangleright$ de la primera celda de cada cinta.

máquinas de k cintas. Podemos pensar que cuando la máquina entra al estado q_f, se apaga, deja de evolucionar.

Así, una **máquina de Turing** (o simplemente **máquina**, o también llamandas **máquinas determinísticas**, para distinguirla de otras que veremos más adelante) M es una tupla (Σ, Q, δ), donde Σ es el alfabeto, Q es el conjunto de estados y δ es la función de transición como fue descripta en (2) y con las restricciones que mencionamos. Notemos que la cantidad de cintas de M se deduce de la signatura de δ.

2.1. Configuraciones y cómputos

Una **configuración** de una máquina $M = (\Sigma, Q, \delta)$ con k cintas es un estado de Q junto con una asignación particular de símbolos del alfabeto a las celdas de cada una de las k cintas, y una asignación particular de posiciones a cada cabeza de M.

La función de transición δ transforma una configuración en otra. Si C y C' son configuraciones, decimos que C **evoluciona en un paso vía** δ a C' si δ transforma C en C'. La Figura 3 muestra una configuración que evoluciona en un paso en otra vía δ.

La **configuración inicial** para $x \in (\Sigma \setminus \{\triangleright\})^*$ de la máquina $M = (\Sigma, Q, \delta)$ con k cintas es una en la que:

- la cinta de entrada contiene $\triangleright$ seguido de x (un símbolo de x en cada celda) y el resto de las celdas en blanco (recordar que x no contiene al símbolo $\triangleright$)
- todas las cintas de trabajo empiezan con $\triangleright$ y tienen el resto de las celdas en blanco
- la cinta de salida empieza con $\triangleright$ y el resto de las celdas en blanco
- todas las cabezas están apuntando a la segunda celda, es decir a la celda siguiente a la que contiene al símbolo $\triangleright$
- el estado es q_0 (estado inicial)

Una **configuración final** de la máquina $M = (\Sigma, Q, \delta)$ con k cintas es cualquiera de esta forma:

- el estado es q_f (estado final); recordemos que una vez que entra a q_f, M se congela, es decir, queda en esa configuración para siempre
- las cintas de trabajo o de salida pueden contener cualquier secuencia de símbolos; por nuestras restricciones de δ los símbolos iniciales $\triangleright$ siempre quedarán escritos en la primera celda de cada cinta
- las cabezas pueden estar en cualquier posición
- si en la cinta de salida hay escrito un $\triangleright 1$ seguido de blancos decimos que la configuración final es **aceptadora**

Decimos que una máquina en una configuración final paró o terminó. La Figura 4 muestra una configuración inicial y una configuración final.

Una máquina $M = (\Sigma, Q, \delta)$ transforma una entrada x –que es lo que tiene escrito en la cinta de entrada inicialmente– en una salida –que es lo que queda escrito en la cinta de salida al entrar al estado final q_f–, o bien nunca entra en el estado final q_f. La entrada x puede ser cualquier palabra formada por símbolos de Σ, excepto $\triangleright$, que sirve para marcar el inicio de las cintas. Fijemos un alfabeto finito Γ tal que $\triangleright \notin \Gamma$ y definamos $\Gamma' = \Gamma \cup \{\triangleright, \square\}$.

Definición 1. Un **cómputo** de $M = (\Gamma', Q, \delta)$ a partir de $x \in \Gamma^*$ es una secuencia $C_0, \ldots, C_\ell$ de configuraciones tal que C_0 es inicial para x, C_ℓ es final y C_{i+1} es la evolución de C_i en un paso vía δ para todo $i \in \{0, \ldots, \ell - 1\}$. Llamamos **longitud** del cómputo a ℓ. Decimos que M con entrada x **termina** o **para** si existe un cómputo de M a partir de x.

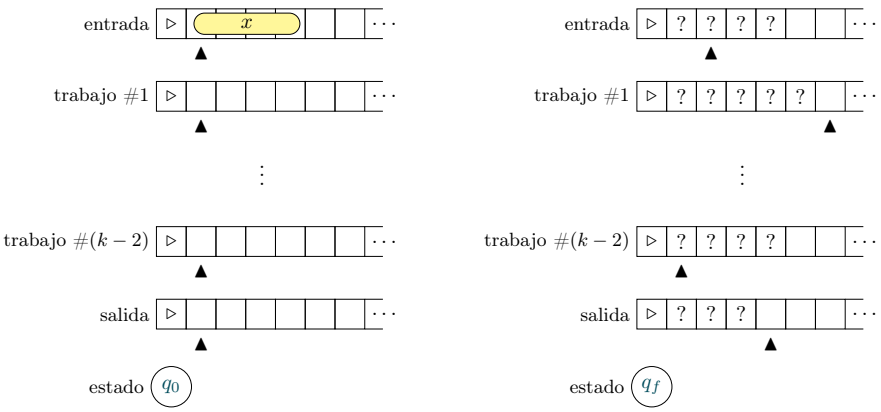


Figura 4: Izquierda: La configuración inicial de una máquina $M = (\Sigma, Q, \delta)$ con k cintas para x. Derecha: una posible configuración final (cada símbolo '?' representa algún símbolo de Σ).

.

2.2. Funciones computables

Hasta aquí, nos dedicamos a definir un modelo de cómputo. La definición que dimos es suficientemente formal para nuestros objetivos. Para definir más formalmente el funcionamiento de una máquina, deberíamos codificar las configuraciones y especificar de qué manera la función de transición transforma una configuración en otra mediante reglas de reescritura basadas en esa codificación. No lo vamos a hacer ahora porque no necesitamos tanta formalidad. Sin embargo, en §4.4 volveremos sobre esta idea de codificación de configuraciones y función de transición.

En esta sección vamos a definir formalmente la noción de método efectivo. Como antes, fijemos un alfabeto finito Γ tal que $\triangleright \notin \Gamma$ y definamos $\Gamma' = \Gamma \cup \{\triangleright, \square\}$.

Definición 2. Sea $f : \Gamma^* \to \Gamma^*$ y sea $M = (\Gamma', Q, \delta)$ una máquina. Decimos que M **computa** f si para todo $x \in \Gamma^*$, hay un cómputo $C_0, \ldots, C_\ell$ de M a partir de x y en C_ℓ la cinta de salida tiene escrito $\triangleright$ seguido de $f(x)$ (un símbolo en cada celda) seguido de celdas en blanco. En este caso, notamos $M(x)$ para representar a lo que tiene escrito M en su cinta de salida cuando entra al estado final q_f, que en este caso es $f(x)$. Decimos que f es **computable**[4] si existe una máquina que la computa.

Notemos que, por el funcionamiento de las máquinas, la cabeza de la cinta de salida siempre estará en una celda en blanco, delimitando así el contenido de $f(x)$. La Figura 5 muestra el cómputo de una máquina que computa f.

Vamos a trabajar con una clase de funciones algo distintas a las que estamos acostumbrados. Se llaman 'funciones parciales'.

Definición 3. Una **función parcial** es una función que puede indefinirse en algunos puntos. Notamos $f(x) \uparrow$ cuando f está indefinida en x y $f(x) \downarrow$ cuando f está definida en x. Llamamos **dominio** de f al conjunto

$$\mathrm{dom} f = \{x \colon f(x) \downarrow\}.$$

A las funciones parciales f sobre Γ^* las vamos a notar $f :\subseteq \Gamma^* \to \Gamma^*$ para marcar que el dominio de f no necesariamente es Γ^*.

Finalmente, llegamos a la definición formal de función parcial computable:

Definición 4. Sea $f :\subseteq \Gamma^* \to \Gamma^*$ una función parcial y sea $M = (\Gamma', Q, \delta)$ una máquina. Decimos que M **computa** f si para todo $x \in \Gamma^*$:

[4] En los libros más viejos se habla de función *recursiva* en vez de *computable*; el término que más se usa ahora es *computable*.

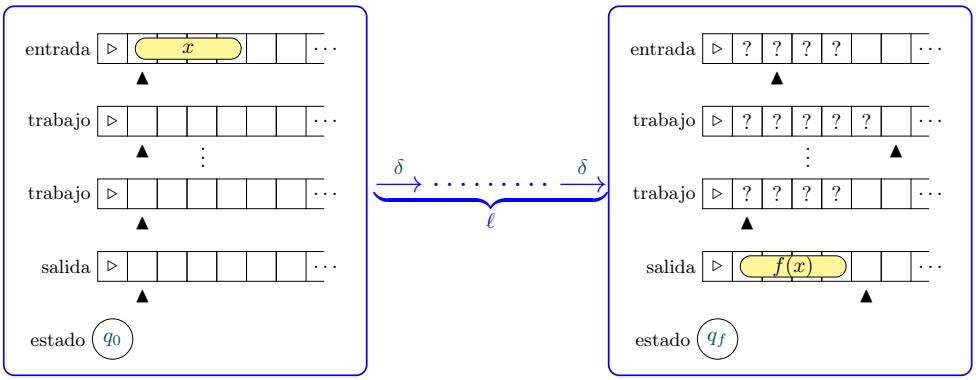


Figura 5: La función f es computable por una máquina. Para cualquier $x \in \Gamma^*$ hay un cómputo a partir de x que termina en una configuración final que tiene a $f(x)$ escrito en la cinta de salida.

- si $f(x) \downarrow$ entonces hay un cómputo $C_0, \ldots, C_\ell$ de M a partir de x y en C_ℓ la cinta de salida tiene escrito $\triangleright$ seguido de $f(x)$ (un símbolo en cada celda) seguido de celdas en blanco. Decimos que $M(x) \downarrow$.
- si $f(x) \uparrow$ entonces hay una secuencia infinita de configuraciones $C_0, C_1, \ldots$ tal que C_0 es inicial C_{i+1} es la evolución de C_i en un paso vía δ para todo $i \geq 0$ y ningún C_i es final. En este caso no hay un cómputo de M a partir de x. En otras palabras, M 'se cuelga' con entrada x.

Decimos que f es **parcial computable** si existe una máquina que la computa.

La necesidad de trabajar con funciones parciales viene del hecho de que, en general, las máquinas pueden **colgarse** (es decir, no terminar). Esta no-terminación de las máquinas se refleja como una indefinición de la función que están computando. Toda máquina computa una función parcial. Cuando la función parcial f está definida para toda entrada, es decir, cuando $\mathrm{dom} f = \Gamma^*$, decimos que f es **total**. Así, las funciones totales son las funciones matemáticas que ya conocíamos. Ahora las extendemos permitiendo que se indefinan en algunos puntos (tal vez en todos, tal vez en ninguno). Las funciones con las que trabajaremos en general serán totales. Sin embargo, el modelo de cómputo son las máquinas y en ese sentido es importante tener en cuenta que en principio nuestro modelo de cómputo describe funciones parciales.

A veces conviene representar a las máquinas $M = (Q, \Sigma, \delta)$ con k cintas en forma de autómatas. Recordemos de (2) que δ mapea tuplas de la forma

$$(q, a, b_1, \ldots, b_{k-1})$$

a tuplas

$$(E, t_1, \ldots, t_{k-2}, T_1, \ldots, T_{k-2}, Z, q')$$

donde E, $T_1, \ldots T_{k-2}$ y Z representan las acciones de la cinta de entrada, las de cada una de las $k-2$ cintas de trabajo y la de salida, respectivamente, $a, b_1, \ldots, b_{k-1}, t_1, \ldots, t_{k-2} \in \Sigma$ y $q, q' \in Q$. Cuando representamos a $M = (Q, \Sigma, \delta)$ en forma de autómatas, los estados del autómata van a ser los estados de Q y la transición entre el estado q y q' está etiquetado por

$$a, b_1, \ldots, b_{k-2} \to E, t_1, \ldots, t_{k-2}, T_1, \ldots, T_{k-2}, Z.$$

A veces no nos va a importar representar algunos valores de $\delta(q, a, b_1, \ldots, b_{k-2})$, porque van a ser irrelevantes para la función que queremos calcular. En este caso, podemos acordar que $\delta(q, a, b_1, \ldots, b_{k-2}) = (S, b_1, \ldots, b_{k-2}, S, \ldots, S, S, r)$, donde r es un estado 'trampa', uno del que no puede salir, es decir, tal que $\delta(r, \ldots) = (\ldots, r)$. En el autómata, ni r ni la correspondiente transición entre q y r se representan.

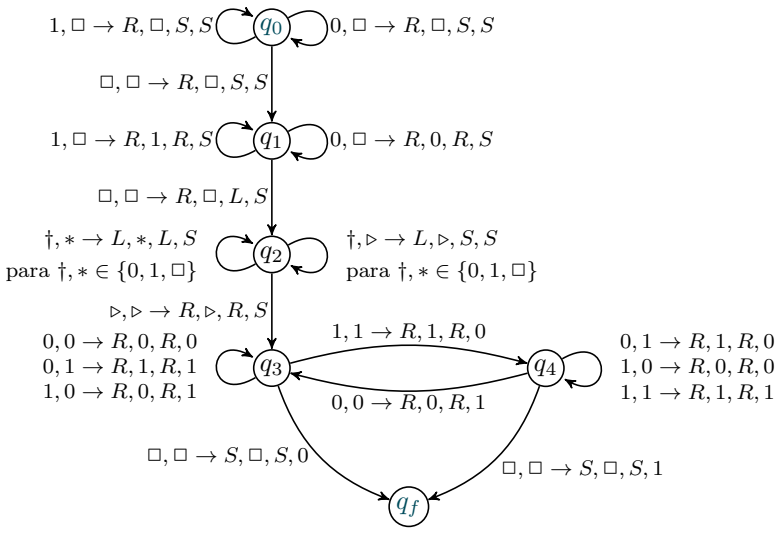


Figura 6: Una máquina representada en forma de autómata tal que, con entrada $x\square y$ ($x, y \in \{0,1\}^n$), devuelve $x + y$, de longitud $n + 1$, representando a los números en binario con el dígito menos significativo a la izquierda.

.

Ejemplo 1. Consideremos la función $f : \subseteq \{0, 1, \square\} \to \{0, 1\}^*$ tal que para para todo $n > 0$ y $x, y \in \{0, 1\}^n$, $f(x\square y) = z$ sii $z = x + y$ y $|z| = n + 1$, si identificamos números naturales con cadenas en $\{0, 1\}^*$ con el dígito menos significativo a la izquierda (o sea, al revés de como los representamos usualmente). Si w no es de la forma $x\square y$ para algún $n > 1$ y $|x| = |y|$, entonces $f(w) \uparrow$. La Figura 6 muestra una máquina M con 3 cintas que computa f.

Supongamos que la entrada es $x\square y$ con $|x| = |y| > 0$. Primero, q_0 lee los símbolos de x hasta llegar al $\square$. Luego q_1 lee los símbolos de y copiándolos, al mismo tiempo, en la cinta de trabajo. A continuación, q_2 retrocede las cabezas de entrada y trabajo y, al entrar a q_3, cada cabeza queda apuntando a la segunda celda de cada cinta, exactamente como al inicio del cómputo. En este punto, la máquina tiene $x\square y$ en la cinta de entrada (aunque solo le va a interesar x) e y en la cinta de trabajo. Ahora la máquina va a sumar $x + y$ bit a bit y va a dejar el resultado en la cinta de salida: q_3 maneja el caso sin *carry* y q_4 el caso con *carry*. Finalmente, llega a q_f escribiendo un 0 o un 1 final en la cinta de salida, según viniera desde q_3 o q_4. La Figura 7 muestra algunas configuraciones del cómputo de M en el cómputo con entrada $10\square 11$ y salida 001.

2.3. Tiempo de cómputo

Como siempre, fijemos un alfabeto finito Γ tal que $\triangleright \notin \Gamma$ y definamos $\Gamma' = \Gamma \cup \{\triangleright, \square\}$.

Definición 5. Sean $f : \Gamma^* \to \Gamma^*$, $T : \mathbb{N} \to \mathbb{N}$ y sea $M = (\Gamma', Q, \delta)$ una máquina. Decimos que M **corre en tiempo** $T(n)$ si para todo $x \in \Gamma^*$ hay un cómputo de M a partir de x de longitud a lo sumo $T(|x|)$ (en particular, M no se cuelga nunca). Decimos que M computa f **en tiempo** $T(n)$ si M corre en tiempo $T(n)$ y M computa f. Ver Figura 8.

En general no vamos a estar interesados en medir exactamente la cantidad de pasos que toma una máquina, sino en una aproximación a grandes rasgos y en el límite. Esta noción se la conoce como notación de 'O grande'.

Definición 6. Sean $f, g : \mathbb{N} \to \mathbb{N}$. Decimos que $f = O(g)$ (se lee 'f es O grande de g') si existe c tal que para todo n suficientemente grande tenemos $f(n) \leq c \cdot g(n)$.

Ejemplo 2. Algunos ejemplos de O grande: $4(n + 2) = O(n)$, $1000000n + 1000000 = O(n)$, $n \log n = O(n^2)$, $5n^2 + 3n + 1 = O(n^2)$, $2^n \neq O(n^k)$ para ningún k.

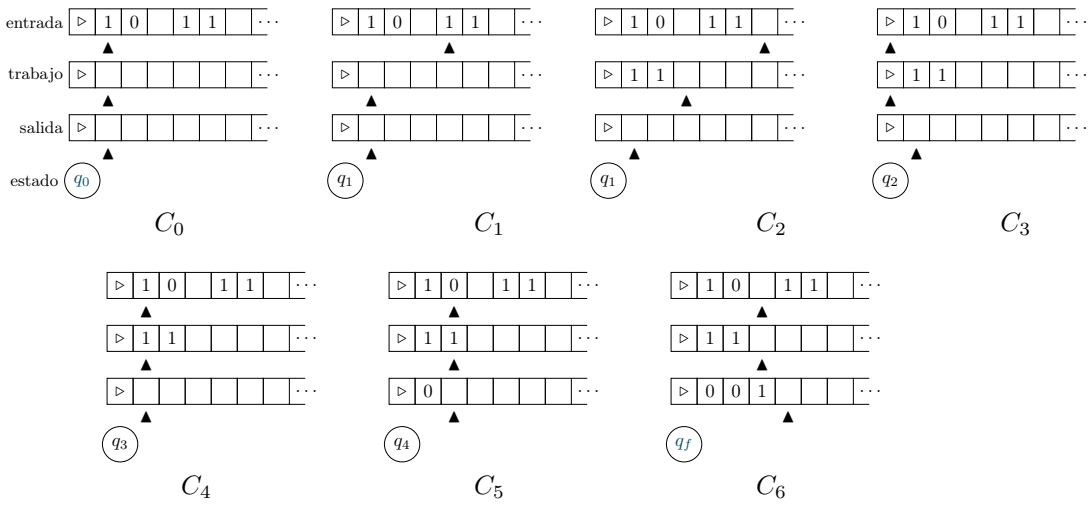


Figura 7: Algunas configuraciones del cómputo de la máquina de la Figura 6 con entrada 10□11: $C_0 \xrightarrow{\delta^*} C_1 \xrightarrow{\delta^*} C_2 \xrightarrow{\delta^*} C_3 \xrightarrow{\delta^*} C_4 \xrightarrow{\delta^*} C_5 \xrightarrow{\delta^*} C_6$, donde δ^* representa varias evoluciones en un paso vía δ, C_0 es la configuración inicial para la entrada 10□11 y C_6 es una configuración final en donde la salida es 001.

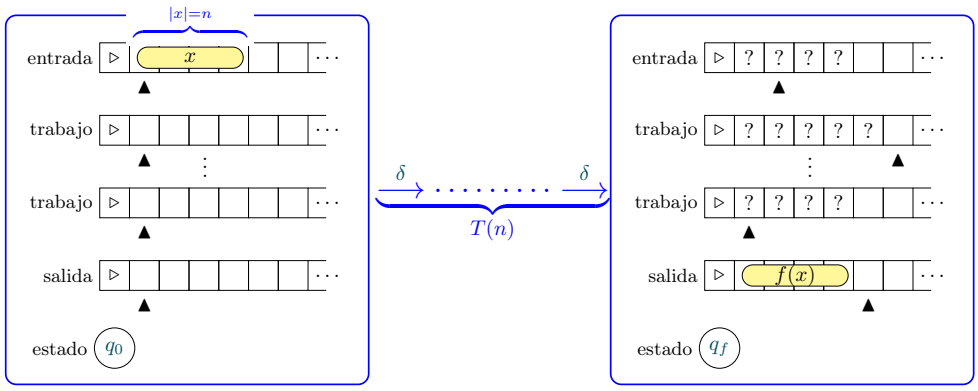


Figura 8: M computa f en tiempo $T(n)$.

Ahora que tenemos una noción de aproximación por O grande, la trasladamos a la cantidad de pasos de una máquina:

Definición 7. Sea $T : \mathbb{N} \to \mathbb{N}$ y $M = (\Gamma', Q, \delta)$ una máquina. Decimos que M **corre en tiempo** $O(T(n))$ si existe una constante c tal que para todo $x \in \Gamma^*$, salvo finitos, hay un cómputo de M a partir de x de longitud a lo sumo $c \cdot T(|x|)$.

Es importante remarcar que el tiempo que toma la máquina en computar se hace en función del *tamaño* de la entrada y no en del valor de la entrada —recordemos, además, que la entrada es una cadena de símbolos sobre el alfabeto Γ, de modo que no tiene un valor numérico dado a priori.

Definición 8. Decimos que M **corre en tiempo polinomial** si existe un polinomio $p(n)$ tal que M corre en tiempo $p(n)$.

Notemos que el hecho de que M corra en tiempo polinomial es equivalente a decir que existe una constante c tal que M corre en tiempo $O(n^c)$.

Hasta ahora explicamos cómo medir el tiempo en los cómputos de *máquinas*. Ahora vamos a ver cómo clasificar a las *funciones* y *lenguajes*.

Definición 9. Sea $f : \Gamma^* \to \Gamma^*$. Decimos que f es **computable en tiempo** $T(n)$ [**computable en tiempo** $O(T(n))$] si existe una máquina $M = (\Gamma', Q, \delta)$ tal que M computa f y M corre en tiempo $T(n)$ [corre en tiempo $O(T(n))$]. Decimos que f es **computable en tiempo polinomial** si existe una máquina $M = (\Gamma', Q, \delta)$ tal que M computa f y M corre en tiempo polinomial.

Decimos que un lenguaje $\mathcal{L}$ es **decidible en tiempo** $T(n)$ [**decidible en tiempo** $O(T(n))$, **decidible en tiempo polinomial**] si $\chi_{\mathcal{L}}$ es computable en tiempo $T(n)$ [computable en tiempo $O(T(n))$, computable en tiempo polinomial].

2.4. Codificación de máquinas

Las máquinas que consideremos de aquí en más van a trabajar con el alfabeto $\Sigma = \{0, 1, \triangleright, \square\}$, al que llamamos **alfabeto estándar**. Las entrada y la salida siempre va a estar formada por cadenas en $\{0, 1\}^*$.

Consideremos una máquina $M = (\Sigma, Q, \delta)$, donde Q es un conjunto finito de estados y $\Sigma = \{0, 1, \triangleright, \square\}$ es el alfabeto estándar. Recordemos que la función de transición es una función como la descripta en (2):

$$\delta : Q \times \Sigma^{k-1} \to \underbrace{\{L, R, S\}}_{\text{entrada}} \times \underbrace{\Sigma^{k-2} \times \{L, R, S\}^{k-2}}_{\text{trabajo}} \times \underbrace{(\Sigma \cup \{S\})}_{\text{salida}} \times Q.$$

Vamos a ver cómo podemos codificar máquinas, es decir asignarle una cadena de $\{0, 1\}^*$ a cada máquina. Tenemos que ir codificando de a poco.

- **Codificación de estados.** Numeramos los estados de Q desde 0 hasta $|Q| - 1$ y representamos cada estado n con $[n]$. Reservamos el 0 para q_0 y el 1 para q_f.
- **Codificación del alfabeto y las acciones**. Codificamos cada símbolo de $\Sigma \cup \{L, R, S\}$ con $[0] = 000$, $[1] = 001$, $[\square] = 010$, $[\triangleright] = 011$, $[L] = 100$, $[R] = 101$, y $[S] = 110$.
- **Codificación de la función de transición δ.** Codificamos una tupla $\vec{v}$

  $$(E, t_1, \ldots, t_{k-2}, T_1, \ldots, T_{k-2}, Z, q) \in \{L, R, S\} \times \Sigma^{k-2} \times \{L, R, S\}^{k-2} \times (\Sigma \cup \{S\}) \times Q$$

  con

  $$\langle \vec{v} \rangle = \langle [E], [t_1], \ldots, [t_{k-2}], [T_1], \ldots, [T_{k-2}], [Z], [q] \rangle \in \{0, 1\}^*$$

y codificamos

$$\delta : Q \times \Sigma^{k-1} \to \{L, R, S\} \times \Sigma^{k-2} \times \{L, R, S\}^{k-2} \times (\Sigma \cup \{S\}) \times Q$$

con

$$\langle \delta \rangle = \langle \{ \langle [q], [a_1], \ldots, [a_{k-1}], \langle \delta(q, a_1, \ldots, a_{k-1}) \rangle \rangle \mid q \in Q, a_1, \ldots, a_{k-1} \in \Sigma \} \rangle \in \{0,1\}^*.$$

Finalmente, codificamos $M = (\Sigma, Q, \delta)$ con k cintas ($k \geq 3$) con

$$\langle M \rangle = \langle [|Q|], [k], \langle \delta \rangle \rangle \in \{0,1\}^*.$$

Si $x \in \{0,1\}^*$ no respeta el patrón $\langle M \rangle$ para alguna $M = (\Sigma, Q, \delta)$, entonces suponemos que representa una máquina trivial fija cualquiera, por ejemplo una que termina ni bien empieza y devuelve la palabra vacía para cualquier entrada. Así, toda palabra $x \in \{0,1\}^*$ representa alguna máquina. Identificamos máquinas con palabras $x \in \{0,1\}^*$ y podemos hablar de 'la máquina x' o 'la x-ésima máquina' para referirnos a la única máquina M tal que $\langle M \rangle = x$. Observemos que toda máquina M representa una función parcial y para toda función parcial f existen infinitas máquinas que computan f. Esto último vale porque dada una máquina M que computa f, podemos variar espuriamente M (por ejemplo, agregando estados que no toman ningún rol en el programa) para obtener otra máquina M' que también computa f. M y M' tendrán codificaciones distintas, es decir $\langle M \rangle \neq \langle M' \rangle$, pero tanto M como M' computan f. De esta forma, toda función parcial se codifica con infinitas palabras en $\{0,1\}^*$.

Como toda máquina se codifica con una palabra de $\{0,1\}^*$, y este conjunto es numerable, entonces el conjunto de funciones (parciales) computables también es numerable. Como el conjunto de funciones $\{0,1\}^* \to \{0,1\}^*$ es no numerable, tiene que haber (una cantidad no numerable de) funciones no computables. ¿Qué ejemplo concreto conocemos? En §2.6 vamos a contestar esta pregunta.[5]

2.5. Variantes de máquinas

El modelo de máquina que definimos es solo una posible forma de definir el dispositivo de cómputo. ¿Por qué trabajamos el alfabeto es el estándar y no permitimos otro con más símbolos? ¿Por qué permitimos cintas que solo son infinitas hacia la derecha? ¿Podríamos trabajar con cintas infinitas en las dos direcciones? Son todas preguntas válidas. No hay una única manera de formalizar la noción de método efectivo pero tenemos que fijar uno para poder desarrollar la teoría. Los resultados de esta sección muestran que las distintas variantes de máquinas son poco importantes para las preguntas que se hace la teoría de la complejidad. Por eso, vamos a poder usar uno u otro según nos convenga.

Definición 10. Una función $T : \mathbb{N} \to \mathbb{N}$ es **construible en tiempo** si $T(n) \geq n$ y la función $1^n \mapsto [T(n)]$ es computable en tiempo $O(T(n))$.

Ejemplo 3. Algunos ejemplos de funciones construibles en tiempo son: $n \log n$, n, n^2, 2^n.

La siguiente Proposición se puede probar fácilmente codificando cada símbolo de Γ en binario usando $\lceil \log |\Gamma| \rceil$ celdas.

Proposición 1. *Sea $f : \{0,1\}^* \to \{0,1\}^*$, sea T una función construible en tiempo y sea Γ un alfabeto. Si f es computable en tiempo $T(n)$ por una máquina $M = (\Gamma, Q, \delta)$, entonces f es computable en tiempo $O(\log |\Gamma| \cdot T(n))$ por una máquina $M' = (\Sigma, Q', \delta')$ donde $\Sigma = \{0, 1, \triangleright, \square\}$ es el alfabeto estándar.*

[5] No es tanta sorpresa: ya lo vieron en *Lenguajes Formales, autómatas y computabilidad.*

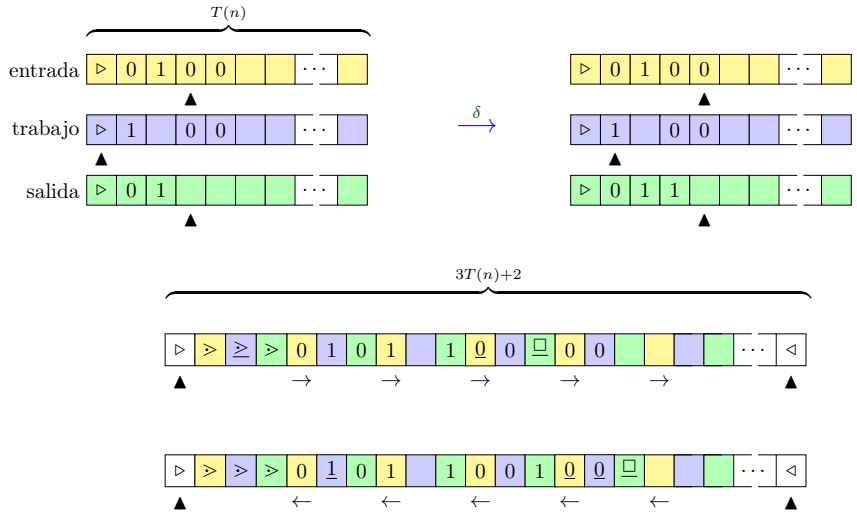


Figura 9: Arriba: la evolución en un paso de M. Abajo: la simulación de M por una máquina de cinta única M', con alfabeto $\Gamma = \{\triangleright, 0, 1, \square, \underline{\triangleright}, \underline{0}, \underline{1}, \underline{\square}, \triangleleft\}$. Los símbolos subrayados indican que la cabeza correspondiente está leyéndolos en la simulación de M llevada adelante por M'. Primero barre de izquierda a derecha y después vuelve barriendo toda la cinta de derecha a izquierda.

Las máquinas **de cinta única** tienen una sola cinta con una cabeza de lectura y escritura. Para estas máquinas, la función de transición tiene la forma

$$\delta : Q \times \Sigma \to \Sigma \times \{L, R, S\} \times Q$$

con restricciones para no pasarse del comienzo de cinta. La entrada se codifica como lo hacemos con las máquinas que venimos estudiando y la salida es lo que queda escrito al principio de la (única) cinta hasta la (única) cabeza.

Proposición 2. *Sea $f : \{0,1\}^* \to \{0,1\}^*$ y sea T una función construible en tiempo. Si f es computable en tiempo $T(n)$ por una máquina estándar de $k \geq 3$ cintas (entrada, salida y $k-2$ cintas de trabajo), entonces f es computable en tiempo $O(T(n)^2)$ por una máquina de cinta única.*

Idea de la demostración. Codificamos el contenido de las cintas de la máquina M que computa f (sobre el alfabeto estándar $\Sigma = \{0, 1, \triangleright, \square\}$) con k cintas en una máquina M' de cinta única con alfabeto $\Gamma = \{\triangleright, 0, 1, \square, \underline{\triangleright}, \underline{0}, \underline{1}, \underline{\square}\}$. El símbolo subrayado indica que la cabeza correspondiente está leyendo ese símbolo. Como M recorre en tiempo $T(n)$, entonces a lo sumo usa $T(n)$ celdas de cada una de sus cintas. Para codificar las cintas de M, la máquina M' alterna los símbolos de cada cinta (todos los primeros símbolos, en orden, después todos los segundos, en orden, y así siguiendo) en su única cinta. Así, M' codifica toda la información de las cintas de M en las primeras $k \cdot T(n)$ celdas. Sabemos que no va a usar más que eso. Ver Figura 9.

M' con entrada x hace esto:

> *Obtiene $|x| = n$ y calcula $T(n)$. Mueve la cabeza hasta la posición $k \cdot T(n) + 1$ y escribe $\triangleleft$. Esto le va a servir para marcar la parte de la cinta única que va a usar (nunca va a usar la cinta más allá de $\triangleleft$). Luego vuelve a la primera posición.*
>
> *Codifica la configuración inicial de M con entrada x en su cinta (esto incluye a x) usando exactamente $k \cdot T(n)$ celdas.*
>
> *Maneja el estado de M internamente, sin necesidad de tener la información escrita en la cinta (finitos estados)*

Repite lo siguiente hasta que M llegue al estado final:

> *Barre la cinta de izquierda a derecha obteniendo (recordando en estados) los símbolos leídos por cada cabeza (subrayados).*
>
> *Dependiendo de qué hace la función de transición de M con esa información, elige qué hacer con cada cabeza.*
>
> *Barre la cinta de derecha a izquierda volcando esa nueva información (M cambia a lo sumo un símbolo por cada cinta y mueve a lo sumo una cabeza por cada cinta)*

Borra todo lo que hay en su cinta salvo la información de la salida de M con entrada x y la deja al principio de la cinta, con la cabeza en el último bit de $M(x)$.

Se puede ver que M' corre en tiempo $O(T(n)^2)$ y calcula f. Después, podemos simular M' con otra máquina que use el alfabeto estándar. □

Definición 11. Una máquina M es **oblivious**[6] si para cada entrada x y para cada $i \in \mathbb{N}$,

- la posición de las cabezas de las cintas de entrada y trabajo en el i-ésimo paso del cómputo de M con entrada x solo depende de i y de $|x|$ (pero no de x), y
- las funciones que computan esas posiciones a partir de $i, |x|$ son computables en tiempo polinomial.

Proposición 3. *Sea $f : \{0,1\}^* \to \{0,1\}^*$ y sea T una función construible en tiempo. Si f es computable en tiempo $T(n)$ por una máquina estándar entonces hay una máquina oblivious que computa f en tiempo $O(T(n)^2)$.*

Idea de la demostración. Se puede probar como en la demostración de la Proposición 2. Efectivamente, la máquina M' de esa demostración es oblivious. En este caso, la máquina tiene que hacer el mismo truco pero solo con la cinta de entrada y trabajo, y dejar el funcionamiento de la cinta de salida como la de M. Ver Figura 10. □

Las **máquinas con cintas bi-infinitas** tienen cintas infinitas en ambas direcciones en lugar de ser infinitas solo a derecha.

Proposición 4. *Sea $f : \{0,1\}^* \to \{0,1\}^*$ y sea T una función construible en tiempo. Si f es computable por una máquina con cintas bi-infinitas en tiempo $T(n)$, entonces f es computable por una máquina estándar en tiempo $O(T(n))$.*

Idea de la demostración. Podemos 'plegar' cada cinta bi-infinita sobre el alfabeto estándar, codificándola sobre el alfabeto $\{\triangleright\} \cup \{0,1,\square\}^2$. Según qué porción de la cinta bi-infinita esté procesando, usaremos la primera o segunda componente, rebotando sobre $\triangleright$. Luego podemos de nuevo traducir al alfabeto estándar. Ver Figura 11. □

2.6. El problema de la detención y las máquinas universales

Recordemos de §2.4 que hablamos de la 'x-ésima máquina' para referirnos a la única máquina M tal que $\langle M \rangle = x$. Definimos $halt : \{0,1\}^* \to \{0,1\}$ como

$$halt(x) = \begin{cases} 1 & \text{si la } x\text{-ésima máquina con entrada } x \text{ termina} \\ 0 & \text{si no} \end{cases}$$

[6] *Oblivious* en inglés quiere decir que no se da cuenta, ajena, inconsciente. Decidí usar directamente el nombre en inglés.

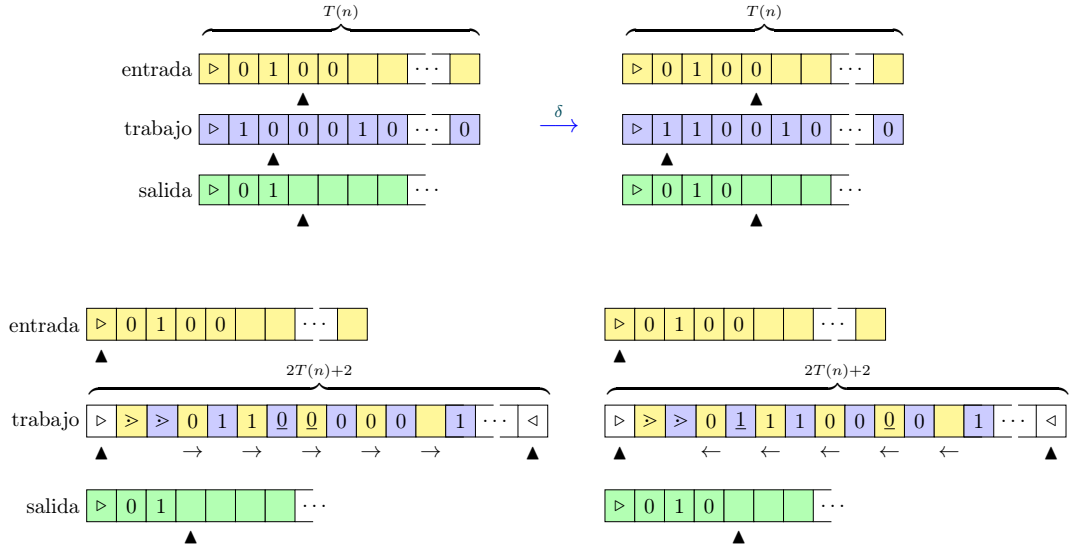


Figura 10: Arriba: la transición en un paso de la máquina M. Abajo: la simulación de la máquina oblivious M'. En una primera instancia, M' copia el contenido de x en la cinta de trabajo, usando las celdas amarillas. Para eso mueve la cabeza de la cinta de entrada. Una vez que copió x en la cinta de trabajo, no vuelve a mover la cabeza de la cinta de entrada y solo mueve la cabeza de la cinta de trabajo y de salida. En la segunda instancia, la cabeza de la cinta de trabajo va barriendo de izquierda a derecha y de derecha a izquierda como en Proposición 2. Las posiciones de las cintas de entrada y trabajo de M' solo dependen de $|x|$ y del número de paso, y se computan polinomialmente a partir de esos valores.

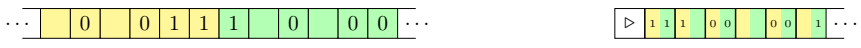


Figura 11: Izquierda: una cinta bi-infinita sobre el alfabeto $\{0, 1, \square\}$. Derecha: el 'plegado' de esa cinta sobre el alfabeto $\{\triangleright\} \cup \{0, 1, \square\}^2$.

Teorema 1. *[Turing 1936] halt no es computable.*

Demostración. Supongamos que *halt* es computable. Definimos una máquina M tal que $M(x)$ termina sii $halt(x) = 0$. Sea $y = \langle M \rangle$. Tenemos que $M(y)$ termina si y solo si la y-ésima máquina con entrada y no termina, y esto ocurre si y solo si $M(y)$ no termina. El absurdo vino de suponer que *halt* era computable. □

Acá tenemos un ejemplo de función no computable y la técnica para construirla se llama *diagonalización*. Esta técnica permite construir un objeto, en este caso la función computada por M, distinto a muchos otros, en este caso todas las funciones parciales computables (M difiere de la x-ésima máquina en el punto x, de ahí el nombre de 'diagonalización'), que es un absurdo. Volveremos sobre la idea de diagonalización en contextos de Complejidad en §5.1, §5.2 y §7.3.

La teoría que estudia las funciones no computables y su grado de incomputabilidad se llama Computabilidad. A esa teoría no le interesan las funciones computables porque son las más simples de todas. En cambio la Teoría de la Complejidad se centra en las funciones computables y no le interesan las que no son computables. Se centra en las computables porque le interesa medir la menor cantidad de recursos (tiempo, pero también espacio) que se necesitan para computarlas.

Llamemos M_i a la i-ésima máquina, es decir, a única máquina M tal que $\langle M \rangle = i$. Definimos $u : \subseteq \{0,1\}^* \to \{0,1\}^*$ de la siguiente forma:

$$u(\langle i, x \rangle) = M_i(x).$$

Observemos que la función u es parcial y en particular $u(\langle i, x \rangle) \downarrow$ si y solo si $M_i(x) \downarrow$. El siguiente teorema muestra que existe una máquina U *universal* en el sentido de que puede simular cualquier máquina.

Teorema 2. *Existe una máquina U que computa la función $u(\langle i, x \rangle) = M_i(x)$. Más aún, si M_i con entrada x termina en t pasos, entonces U con entrada $\langle i, x \rangle$ termina en $c \cdot t \cdot \log t$ pasos, donde c depende solo de i.*

Idea de la demostración para un caso simple. Supongamos que tenemos una máquina $M = (\Sigma, Q, \delta)$ con *una sola* cinta de trabajo. Vamos a definir la máquina U con 3 cintas de trabajo. U recibe como entrada $\langle \langle M \rangle, x \rangle$ y su objetivo es copiar lo que hace M con la entrada x. Recordemos de §2.4 que $\langle M \rangle$ tiene la información sobre el comportamiento de M, en particular, la función de transición δ. En primer lugar U copia x en la cinta de trabajo #1 y escribe $[q_0]$ en la cinta de trabajo #3. La cinta #2 la reserva para el contenido de la cinta de trabajo de M a medida que la va simulando paso a paso (ver Figura 12 - izquierda). La idea es que mientras que lo que esté escrito en en la cinta #3 no sea (la codificación de) $[q_f]$, U busca en la cinta de entrada la información de δ que necesita y actualiza las cintas #2 y #3 de manera acorde. Si M con entrada x termina en t pasos, esta simulación usa $c \cdot t$ pasos, donde c depende de M pero no de x. Notemos que U tiene que buscar la definición de δ en la cinta de entrada por cada instrucción que simula de M.

Este argumento solo sirve cuando M tiene una sola cinta de trabajo. Es cierto que si tiene dos cintas de trabajo, podemos definir U con tres y seguir la misma idea. Pero necesitamos definir U con una cantidad fija de cintas, porque la definición de U no puede depender de la máquina que esté simulando. Entonces si M tiene más de una cinta de trabajo, se puede transformar en una máquina M' que usa *una sola* cinta con el mismo comportamiento de M. De acuerdo a la Proposición 2, si M llegaba al resultado en tiempo t, M' lo hace en tiempo $O(t^2)$. Esto es demasiado para la cota $O(t \log t)$ del teorema que queremos demostrar. La demostración del caso general es más técnica y no la vamos a estudiar. □

Definamos ahora $\tilde{u} : \{0,1\}^* \to \{0,1\}^*$ como

$$\tilde{u}(\langle i, t, x \rangle) = \begin{cases} 1\ M_i(x) & \text{si } M_i(x) \text{ termina a lo sumo en } t \text{ pasos} \\ 0 & \text{si no} \end{cases}$$

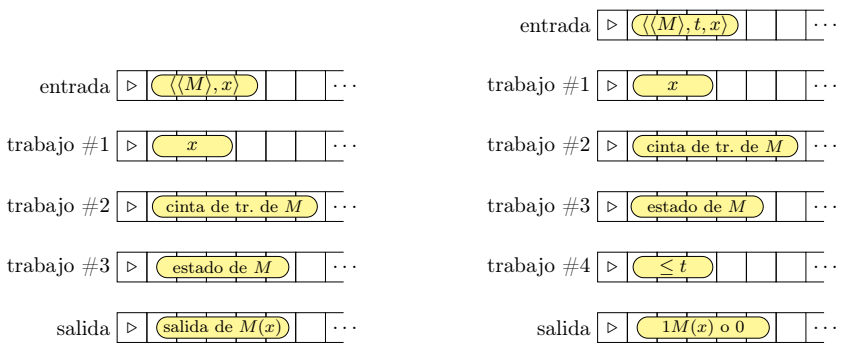


Figura 12: Izquierda: la simulación que hace U (con 3 cintas de trabajo) de M (con una única cinta de trabajo) y entrada x. Derecha: la simulación que hace $\tilde{U}$ (con 4 cintas de trabajo) de M (con una única cinta de trabajo), entrada x hasta el tiempo t.

Observemos que la función $\tilde{u}$ es total y que el primer bit de la salida de $u(\langle i, t, x\rangle)$ nos dice si $M_i(x)$ termina o no en a lo sumo t pasos. En caso de que termine, los bits que siguen son la salida de M_i con entrada x.

Teorema 3. *Existe una máquina $\tilde{U}$ que computa la función $\tilde{u}(\langle i, t, x\rangle)$ en tiempo $c \cdot t \cdot \log t$, donde c depende solo de i.*

Idea de la demostración. Es parecido a la demostración del Teorema 2, pero ahora $\tilde{U}$ tiene una cinta de trabajo más, que usa para llevar cuenta de la cantidad de pasos en la simulación (ver Figura 12 - derecha). □

3. Tiempo polinomial

Empezamos ahora a estudiar una clase de funciones fundamental para la Teoría de la Complejidad: las funciones computables en tiempo polinomial, consideradas como las funciones factibles, las más bondadosas, las eficientemente implementables (veremos más adelante que hay algunas todavía más bondadosas).

Recordemos de §1.3 que un lenguaje $\mathcal{L}$ puede ser visto como una función booleana $\chi_{\mathcal{L}}$, que corresponde a su función característica (ver (1)).

Definición 12. Una máquina $M = (Q, \Sigma, \delta)$ **decide** el lenguaje $\mathcal{L} \subseteq \{0,1\}^*$ [en tiempo $T(n)$] si M computa $\chi_{\mathcal{L}}$ [en tiempo $T(n)$]. Decimos que M **acepta** x cuando $M(x) = 1$ y que M **rechaza** x cuando $M(x) = 0$. Notamos $\mathcal{L}(M)$ al lenguaje decidido por M.

Una **clase de complejidad** es un conjunto de problemas que son decidibles por máquinas con ciertos recursos acotados.

Por ahora, nos centramos en el recurso de *tiempo de cómputo*. Empezamos definiendo la clase de problemas que se resuelven en tiempo T. Como explicamos en §1.3, aquí *resolver* quiere decir computar la función característica del lenguaje inducido por el problema.

Clase de complejidad: DTime$(T(n))$

DTime$(T(n))$ es la clase de lenguajes $\mathcal{L} \subseteq \{0,1\}^*$ tal que existe una máquina $M = (Q, \Sigma, \delta)$ que decide $\mathcal{L}$ en tiempo $O(T(n))$.[7]

[7] Es decir, M computa la función $\chi_{\mathcal{L}}$ en tiempo $O(T(n))$, ver Definición 5.

Por ahora, una 'máquina' $M = (\Sigma, Q, \delta)$ es un dispositivo *determinístico*, es decir, el cómputo evoluciona paso a paso de una única manera, porque δ es una función. De ahí la 'D' en **DTime**. Más adelante, en §3.3, veremos máquinas *no-determinísticas*.

3.1. La clase P

Definimos la clase de problemas que se resuelven en tiempo polinomial respecto al tamaño de su entada. Esta es la formalización más aceptada de la noción de problema *factible* o *eficientemente* resoluble.

Clase de complejidad: P

$$\mathbf{P} = \bigcup_{c>0} \mathbf{DTime}(n^c)$$

Decimos "el problema X está en **P**" o "el problema X se resuelve en tiempo polinomial" para referirnos a que el lenguaje que corresponde a X está en **P**.

Ejemplo 4. Un grafo $G = (V, E)$ es *conexo* si todo par de nodos de G está unido por un camino. Definimos el siguiente problema:

Problema: Conectividad de un grafo

$$\mathsf{CON} = \{\langle G\rangle : G \text{ es un grafo conexo}\}$$

Observemos que CON es un conjunto de cadenas en $\{0,1\}^*$, es decir, un lenguaje. Como ya vimos en §1.2, representamos $G = (V, E)$ en binario con su matriz de adyacencia y suponemos $V = \{0, \dots, k-1\}$. Hay una máquina M que hace esto:

> *Explora G usando* depth first search *desde u y marca los nodos visitados. Si quedó un nodo sin visitar, escribe 0 en la salida; si no, escribe un 1. Luego termina (que quiere decir pasar a q_f).*

Si $n = |\langle G\rangle|$ (en general la letra n va a representar el tamaño de la entrada), M llega al resultado en tiempo $O(n^2)$, entonces $\mathsf{CON} \in \mathbf{P}$.

En realidad, para calcular funciones valuadas en $\{0,1\}$, no hace falta una cinta de salida, porque solo hay dos respuestas posibles: 0 o 1. Podemos, alternativamente, trabajar con dos modelos simplificados de máquinas que solo tienen una cinta de entrada y una trabajo. Un modelo consiste en dejar la salida (0 o 1) en la celda en la que termina la cabeza de trabajo cuando entra a q_f. La otra es reemplazar q_f por un $q_{\text{sí}}$ (que significa que devuelve 1) y un q_{no} (que significa que devuelve 0). En este último modelo el estado de la configuración final es ahora $q_{\text{sí}}$ o q_{no}. Llamemos a cualquiera de estas variantes una máquina **sin cinta de salida**.

El siguiente resultado es trivial.

Proposición 5. *Si $\mathcal{L}$ es decidible en tiempo $T(n)$ por una máquina estándar de 3 cintas (entrada, trabajo y salida), entonces $\mathcal{L}$ es decidible en tiempo en tiempo $O(T(n))$ por una máquina sin cinta de salida.*

3.2. La clase NP

Sabemos que hay una gran diferencia entre escribir una demostración de algún enunciado matemático y verificar que una demostración ya escrita sea correcta. La primera tarea parece tener mucha más creatividad, y, al mismo tiempo, más dificultad. La segunda parece ser más mecánica, porque solo hay que revisar que cada paso de la demostración sea correcto y que finalmente se llegue al resultado deseado. Así como **P** formaliza la clase de problemas eficientemente *resolubles*, la clase **NP** captura la intuición de los problemas eficientemente *verificables*.

Definiremos la clase de problemas que se pueden *verificar* en tiempo polinomial respecto al tamaño de su entada. Observemos que ahora no estamos resolviendo el problema sino verificando que una instancia x dada es una respuesta positiva al problema. El verificador tiene dos entradas: la instancia x del problema y un certificado, que es una explicación, prueba, o justificación de que x es una instancia positiva del problema. El motivo del nombre de esta nueva clase, **NP**, va a quedar claro más adelante.

Clase de complejidad: NP

NP es la clase de lenguajes $\mathcal{L} \subseteq \{0,1\}^*$ tal que existe un polinomio $p : \mathbb{N} \to \mathbb{N}$ y una máquina $M = (Q, \Sigma, \delta)$ tal que M corre en tiempo polinomial y para todo x:

$$x \in \mathcal{L} \quad \text{sii} \quad \text{existe } u \in \{0,1\}^{p(|x|)} \text{ tal que } M(\langle x, u\rangle) = 1. \tag{3}$$

M se llama el **verificador** para $\mathcal{L}$; u se llama **certificado** para x.

Notemos que (3) es equivalente a

$$x \in \mathcal{L} \quad \text{sii} \quad \text{existe } u \in \{0,1\}^{p(|x|)} \text{ tal que } M(xu) = 1 \tag{4}$$

porque $M(y)$ puede primero contar la cantidad m de celdas en la cinta de entrada hasta el primer blanco y luego buscar el primer n tal que $m = n + p(n)$. Tenemos que

$$y(0), \ldots, y(n-1) = x \quad \text{y} \quad y(n), \ldots, y(n + p(n) - 1) = u.$$

Este cálculo inicial de M le toma tiempo polinomial, de modo que podemos usar indistintamente (3) o (4).

Teorema 4. **P** $\subseteq$ **NP**.

Demostración. Supongamos que $\mathcal{L} \in$ **P** y supongamos una máquina M tal que M decide $\mathcal{L}$ en tiempo polinomial. Tomamos $p(n) = 0$. Definimos la máquina M' que, dado $\langle x, \epsilon\rangle$ como entrada, copia el comportamiento de M con entrada x. Como M corre en tiempo polinomial, M' también. Luego,

$$\begin{aligned} x \in \mathcal{L} \quad & \text{sii} \quad M(x) = 1 \\ & \text{sii} \quad M'(\langle x, \epsilon\rangle) = 1 \\ & \text{sii} \quad \text{existe } u \in \{0,1\}^0 \text{ tal que } M'(\langle x, u\rangle) = 1 \end{aligned}$$

y esto concluye la demostración. □

Ejemplo 5. Un conjunto X de nodos de un grafo $G = (V, E)$ es **independiente** si no existen $u, v \in X$ tal que $(u, v) \in E$.

Problema: Conjunto independiente

$$\text{INDSET} = \{\langle G, k\rangle : \ G \text{ tiene un conjunto independiente de } k \text{ o más vértices}\}$$

Veamos que INDSET $\in$ **NP**. Como siempre, supongamos que $V = \{[0], [1], \ldots, [|V| - 1]\}$. Un certificado será una lista de k nodos distintos de V que forman un conjunto independiente. Podemos codificar ese certificado en una palabra u de tamaño $O(k\lceil \log |V| \rceil)$. Sea $n = |\langle G, k\rangle|$. Como $k \leq |V|$, entonces $|u| = O(n \log n)$, usando la codificación de listas. Luego $|u| = p(n)$ para un polinomio cuadrático p (completar con 0s al final). Consideremos la máquina M que con entrada x hace esto:

> *Si x no es de la forma $\langle\langle G, k\rangle, u\rangle$ con $|u| = p(n)$, escribe 0 en la salida. Si no: si u codifica un conjunto independiente de G de k nodos, escribe 1 en la salida; si no, escribe 0.*

Es claro que M corre en tiempo polinomial y además $x \in$ INDSET sii existe $u \in \{0,1\}^{p(|x|)}$ tal que $M(\langle x, u\rangle) = 1$. Entonces INDSET $\in$ **NP**.

3.3. Máquinas no-determinísticas

Las máquinas que vimos hasta ahora son *determinísticas*: cada configuración evoluciona de una única forma a partir de la configuración inicial. Recordemos la Definición 1 de *cómputo*. A partir de ahora, a estas máquinas las vamos a llamar también **máquinas determinísticas**, para distinguirlas de otro tipo de máquinas que vamos a definir a continuación.

Las **máquinas no-determinísticas** son triplas (Σ, Q, δ), como antes, pero con dos diferencias respecto a las determinísticas. La primera es que la **función de transición** es ahora de la forma

$$\delta : Q \times \Sigma^{k-1} \to \left(\underbrace{\{L, R.S\}}_{\text{entrada}} \times \underbrace{\Sigma^{k-2} \times \{L, R, S\}^{k-2}}_{\text{trabajo}} \times \underbrace{(\Sigma \cup \{S\})}_{\text{salida}} \times Q \right)^2 .$$

Observemos la diferencia con la función de transición de una máquina determinística dada en (2): ahora δ especifica una o dos posibles **evoluciones en un paso** a partir de una configuración dada. La segunda diferencia es que es que Q tiene 3 estados distinguidos: q_0, $q_{\text{sí}}$, y q_{no}.

Definición 13. Una **configuración final** de una máquina no-determinística $N = (\Sigma, Q, \delta)$ es una en la que el estado es $q_{\text{sí}}$ o q_{no}. Un **cómputo** de N a partir de $x \in \{0,1\}^*$ es una secuencia $C_0, \ldots, C_\ell$ de configuraciones tal que C_0 es inicial a partir de x, para todo $i \in \{0, \ldots, \ell - 1\}$, C_{i+1} es la evolución de C_i en un paso dado por alguna de las 2 tuplas de δ y C_ℓ es una configuración final. Un cómputo **aceptador** es uno en el que C_ℓ está en estado $q_{\text{sí}}$, y en ese caso a C_ℓ lo llamamos configuración **aceptadora**. Llamamos **longitud** del cómputo a ℓ.

Comparemos esta última definición con la Definición 1. A diferencia de las máquinas determinísticas, ahora tenemos posiblemente muchos cómputos a partir de una configuración dada. En cambio, las máquinas determinísticas tienen a lo sumo un cómputo.

Definición 14. Decimos que la máquina no-determinística $N = (\Sigma, Q, \delta)$ **acepta** x si existe un cómputo aceptador $C_0, \ldots, C_\ell$ de N a partir de $x \in \{0,1\}^*$. En caso contrario decimos que N **rechaza** x. Decimos que N **decide** el lenguaje $\mathcal{L}$ si para todo $x \in \{0,1\}^*$, tenemos que $x \in \mathcal{L}$ sii N acepta x. Notamos $N(x) = 1$ si N acepta x y $N(x) = 0$ si N rechaza x. Notamos $\mathcal{L}(N)$ al lenguaje decidido por N.

Observemos que las máquinas no-determinísticas no computan funciones; solo deciden lenguajes, que, como vimos en §1.3 es una forma de computar funciones valuadas en $\{0,1\}$. En las máquinas no-determinísticas la cinta de salida es irrelevante, porque no participa de ninguna definición. La salida para un cómputo particular viene codificada en el estado $q_{\text{sí}}$ o q_{no}

Definición 15. La máquina no-determinística N **corre en tiempo** $T(n)$ si para todo $x \in \{0,1\}^*$, *todo* cómputo de N a partir de x tiene longitud a lo sumo $T(|x|)$. La máquina no-determinística N **corre en tiempo** $O(T(n))$ si existe una constante c tal que para todo $x \in \{0,1\}^*$, salvo finitos, *todo* cómputo de N a partir de x tiene longitud a lo sumo $c \cdot T(|x|)$. Decimos que N **corre en tiempo polinomial** si existe un polinomio p tal que N corre en tiempo p.

Clase de complejidad: NDTime$(T(n))$

NDTime$(T(n))$ es la clase de lenguajes $\mathcal{L} \subseteq \{0,1\}^*$ tal que existe una máquina no-determinística $N = (Q, \Sigma, \delta)$ tal que N decide $\mathcal{L}$ y N corre en tiempo $O(T(n))$.

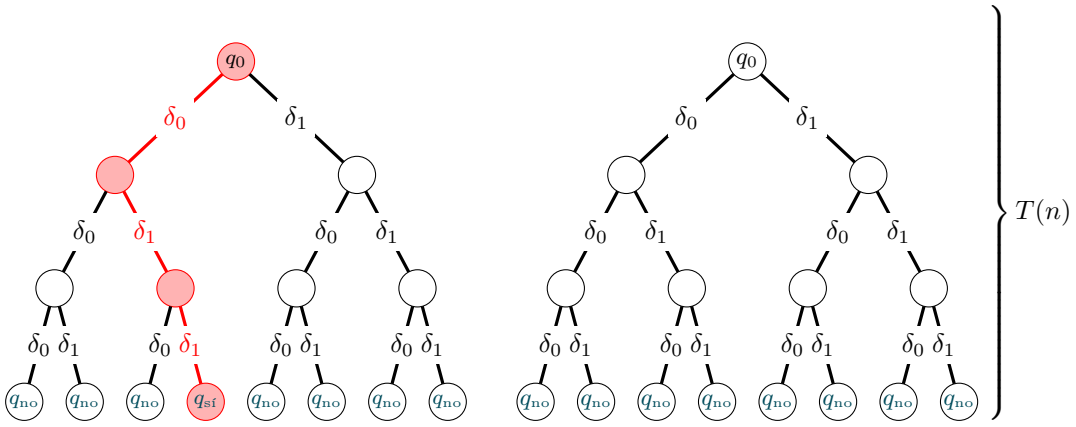


Figura 13: Los cómputos de una máquina no-determinística N que corre en tiempo $T(n)$. Los nodos de cada árbol representan configuraciones de N. La raíz, etiquetada con q_0, representa la configuración inicial de N para una entrada x de tamaño n. Si $\delta(q, \dots) = (a, b)$, definamos $\delta_0(q, \dots) = a$ y $\delta_1(q, \dots) = b$. Si un nodo interno representa a la configuración C, entonces sus hijos corresponden a las dos posibles evoluciones en un paso a partir de C dadas por δ. Las hojas están etiquetadas con el estado correspondiente a la configuración. Izquierda: N acepta x porque existe un cómputo aceptador; 011 es un cómputo aceptador de $N(x)$. Derecha: N rechaza x porque no existe un cómputo aceptador; 011 ni ningún otro cómputo de $N(x)$ es aceptador.

.

El prefijo 'ND' en el nombre de esta clase viene de 'no-determinístico'.

Veamos que podemos codificar todos los cómputos de una máquina no-determinística $N = (\Sigma, Q, \delta)$ con entrada x que corre en tiempo $T(n)$ como un árbol binario $T_{N,x}$ de altura $T(n)$. Cada camino en el árbol desde la raíz corresponde a un cómputo. $T_{N,x}$ se define de esta manera: los nodos del árbol están etiquetados con configuraciones de N con entrada x. La raíz corresponde a la configuración inicial de N con entrada x. Para cada nodo interno etiquetado con C, sus hijos están etiquetados con C_1 y C_2, que son las dos dos posibles evoluciones en un paso a partir de C dadas por δ (puede haber varios nodos del árbol etiquetados con la misma configuración). Si en $T_{N,x}$ existe una hoja etiquetada con la configuración final entonces N acepta x; si no, N rechaza x. En §21 desarrollaremos más esta idea, pero utilizando un grafo y probaremos un resultado más general.

Notemos que podemos codificar los cómputos como con secuencias $\{0,1\}^{T(n)}$, donde 0 significa 'ir al hijo izquierdo' y 1 significa 'ir al hijo derecho'. La Figura 13 (izquierda) muestra en rojo un cómputo aceptador de N. La Figura 13 (derecha) muestra que N rechaza x.

Sea $N = (\Sigma, Q, \delta)$ una máquina no-determinística. Podemos simular determinísticamente *todos* los cómputos de longitud a lo sumo t de N a partir de un x dado y determinar cuáles de esos cómputos son aceptadores. Como a cada cómputo lo codificamos como una cadena en $\{0,1\}^t$, hay 2^t cómputos para analizar. Una vez que fijamos un cómputo representado por $u \in \{0,1\}^t$, podemos simular N como si fuera determinística, solo que cuál de las componentes de δ tenemos que tomar en cada paso viene dado por u. Entonces, una vez que tenemos un u fijo, simulamos N para el cómputos u en $O(t)$ pasos. Como esto lo tenemos que hacer para cada $u \in \{0,1\}^t$, simular determinísticamente *todos* los cómputos de longitud a lo sumo t nos toma tiempo $O(t \cdot 2^t)$. Así, si N corre en tiempo $T(n)$, entonces podemos determinar si N acepta o rechaza x con una máquina determinística que corre en tiempo $O(T(n) \cdot 2^{T(n)})$, que es también $O(2^{T(n)^2})$. En §6 volveremos sobre este resultado y probaremos algo más general.

3.4. Definición de NP vía máquinas no-determinísticas

El siguiente resultado da una definición alternativa de **NP**. Muestra que las siguientes dos propiedades son equivalentes:

- Existe un polinomio $p : \mathbb{N} \to \mathbb{N}$ y una máquina determinística M que corre en tiempo polinomial tal que para todo $x \in \{0,1\}^*$, $x \in \mathcal{L}$ sii existe $u \in \{0,1\}^{p(|x|)}$ tal que $M(\langle x, u\rangle) = 1$
- Existe una máquina no-determinística N que corre en tiempo polinomial tal que $\mathcal{L}(N) = \mathcal{L}$ (es decir, $x \in \mathcal{L}$ sii existe un cómputo aceptador de N a partir de x)

Esta equivalencia se desprende del siguiente teorema:

Teorema 5. $\mathbf{NP} = \bigcup_{c\in\mathbb{N}} \mathbf{NDTime}(n^c)$.

Demostración. Veamos que $\mathbf{NP} \supseteq \bigcup_{c\in\mathbb{N}} \mathbf{NDTime}(n^c)$. Tomemos $\mathcal{L} \in \bigcup_{c\in\mathbb{N}} \mathbf{NDTime}(n^c)$ y veamos que $\mathcal{L} \in \mathbf{NP}$. Supongamos que $N = (\Sigma, Q, \delta)$ es una máquina no-determinística y que p es un polinomio tal que N corre en tiempo $p(n)$ y N decide $\mathcal{L}$, es decir, $x \in \mathcal{L}$ sii existe un cómputo aceptador $C_0, \ldots, C_{p(|x|)}$ de N a partir de x.

Ya vimos que podemos codificar cómputos con cadenas binarias. Consideremos $u \in \{0,1\}^{p(|x|)}$ tal que $u(i) = 0$ si se usa la primera componente de δ para pasar de C_i a C_{i+1} y $u(i) = 0$ en caso contrario ($i = 0, \ldots, p(|x|) - 1$).

Existe una máquina determinística M (el verificador) tal que M con entrada $\langle x, u\rangle$ verifica que u (el certificado) sea la codificación de un cómputo aceptador de N a partir de $x \in \{0,1\}^*$. Si lo es, escribe 1 en la salida y si no escribe 0. M simula N con entrada x paso a paso, y en cada iteración i usa la primera o la segunda componente de δ según el valor de $u(i)$ para saber qué acción realizar. Así, tenemos que $x \in \mathcal{L}$ sii existe $u \in \{0,1\}^{p(|x|)}$ tal que $M(\langle x, u\rangle) = 1$. Como M corre en tiempo polinomial, concluimos que $\mathcal{L} \in \mathbf{NP}$.

Veamos ahora que $\mathbf{NP} \subseteq \bigcup_{c\in\mathbb{N}} \mathbf{NDTime}(n^c)$. Tomemos $\mathcal{L} \in \mathbf{NP}$ y veamos que $\mathcal{L} \in \bigcup_{c\in\mathbb{N}} \mathbf{NDTime}(n^c)$. Supongamos una máquina determinística M que corre en tiempo polinomial y un polinomio $p : \mathbb{N} \to \mathbb{N}$ tal que para todo $x \in \{0,1\}^*$, se cumple que $x \in \mathcal{L}$ sii existe $u \in \{0,1\}^{p(|x|)}$ tal que $M(\langle x, u\rangle) = 1$. Definimos una máquina no-determinística N que con entrada x hace esto:

"inventa" una palabra $u \in \{0,1\}^{p(|x|)}$ en su cinta de trabajo
lo logra con el no-determinismo de δ, que escribe 0 o 1. *(tiempo $O(p(|x|))$).*
simula M con entrada $\langle x, u\rangle$ *(tiempo polinomial, Teorema 2).*
si $M(\langle x, u\rangle) = 1$ entonces entra al estado q_{si}; si no entra al estado q_{no}

Tenemos que $x \in \mathcal{L}$ sii existe un cómputo aceptador de N a partir de x. Como N corre en tiempo polinomial, entonces $\mathcal{L} \in \bigcup_{c\in\mathbb{N}} \mathbf{NDTime}(n^c)$. □

3.5. Máquinas no-determinísticas universales

Análoga a la codificación de máquinas determinísticas que vimos en §2.4, existe una codificación con palabras en $\{0,1\}^*$ de las máquinas no-determinísticas, de forma que toda palabra $x \in \{0,1\}^*$ representa alguna máquina no-determinística. Identificamos máquinas no-determinísticas con palabras $x \in \{0,1\}^*$; hablamos de 'la máquina no-determinística x' o 'la x-ésima máquina no-determinística' para referirnos a la única máquina no-determinística N tal que $\langle N\rangle = x$. Si N es una máquina no-determinística, representamos con $\langle N\rangle \in \{0,1\}^*$ al código de N. Entonces, también hay una cantidad numerable de máquinas no-determinísticas. Llamemos N_i a la máquina no-determinística tal que $\langle N\rangle = i$. El siguiente resultado es análogo al Teorema 2.

Teorema 6. *Existe una máquina no-determinística NU que tal que NU acepta $(\langle i, x\rangle)$ sii N_i acepta x y si N_i corre en tiempo $T(n)$ entonces $NU(\langle i, x\rangle)$ decide si N_i acepta o rechaza x en $c \cdot T(|x|)$ pasos, donde c depende solo de i.*

Notar que en el escenario de las máquinas no-determinísticas, la máquina universal es más eficiente que para el escenario de las máquinas determinísticas, porque elimina el factor logarítmico en la cota del tiempo.

4. Reducción polinomial y problemas NP-completos

Vamos a formalizar la noción de que un problema $\mathcal{L}$ sea a lo sumo tan difícil como un problema $\mathcal{L}'$.

Definición 16. $\mathcal{L} \subseteq \{0,1\}^*$ es **Karp reducible polinomialmente** o simplemente **reducible polinomialmente** a $\mathcal{L}' \subseteq \{0,1\}^*$, notado $\mathcal{L} \leq_p \mathcal{L}'$, si existe una función computable en tiempo polinomial $f : \{0,1\}^* \to \{0,1\}^*$ tal que para todo $x \in \{0,1\}^*$

$$x \in \mathcal{L} \qquad \text{sii} \qquad f(x) \in \mathcal{L}'.$$

f se llama **reducción polinomial** de $\mathcal{L}$ a $\mathcal{L}'$. En este caso decimos que $\mathcal{L} \leq_p \mathcal{L}'$ **vía** f.

Entonces, si queremos decidir si $x \in \mathcal{L}$, podemos transformar x a $f(x)$ y decidir si $f(x) \in \mathcal{L}$. La transformación f tiene que ser una función computable en tiempo polinomial. En particular, esto no puede hacer crecer el tamaño de $f(x)$ más que polinomialmente respecto de $|x|$.

Introducimos dos nuevas clases de complejidad:

Clase de complejidad: NP-hard, NP-completo

$\mathcal{L}$ es **NP-hard** si $\mathcal{L}' \leq_p \mathcal{L}$ para todo $\mathcal{L}' \in \mathbf{NP}$
$\mathcal{L}$ es **NP-completo** si $\mathcal{L} \in \mathbf{NP}$ y $\mathcal{L}$ es **NP-hard**

Así, los problemas **NP-hard**[8] son los problemas al menos tan difíciles como los de **NP** y los problemas **NP-completos** son los problemas **NP-hard** que además están en **NP**, es decir, los problemas más difíciles de todos dentro de los de **NP**. Es fácil ver que la clase **P** está cerrado hacia abajo por la reducción $\leq_p$:

Ejercicio 1. Si $\mathcal{L} \leq_p \mathcal{L}'$ y $\mathcal{L}' \in \mathbf{P}$, entonces $\mathcal{L} \in \mathbf{P}$.

Teorema 7. *La relación $\leq_p$ es transitiva.*

Demostración. Supongamos que $\mathcal{L} \leq_p \mathcal{L}'$ vía f y $\mathcal{L}' \leq_p \mathcal{L}''$ vía g, es decir

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

Veamos que $g \circ f$ es computable en tiempo polinomial.

Recordemos que una máquina determinística que corre en tiempo $T(n)$ no puede escribir más que $T(n)$ símbolos en su cinta de salida. Supongamos que M_f es una máquina determinística que computa f en tiempo $O(n^c)$ y que M_g es una máquina determinística que computa g en tiempo $O(n^d)$. Definimos la máquina determinística M con entrada x de esta manera:

Simula M_f con entrada x para obtener un y *(tiempo $O(n^c)$)*
Simula $M_g(y)$ y escribe el resultado en la cinta de salida *($|y| = O(n^c)$, tiempo $O(n^{cd})$)*

Es claro que $M(x) = g(f(x))$, es decir, que M computa $g \circ f$. Así, probamos que $\mathcal{L} \leq_p \mathcal{L}''$ vía $g \circ f$. □

Teorema 8. *Si* **NP-hard** $\cap$ **P** $\neq \emptyset$*, entonces* **P** $=$ **NP**.

Demostración. Sea $\mathcal{L} \in \mathbf{NP\text{-}hard} \cap \mathbf{P}$. Para ver que $\mathbf{NP} \subseteq \mathbf{P}$, fijemos $\mathcal{L}' \in \mathbf{NP}$ cualquiera y veamos que $\mathcal{L}' \in \mathbf{P}$. Como $\mathcal{L} \in \mathbf{NP\text{-}hard}$, tenemos $\mathcal{L}' \leq_p \mathcal{L}$. Entonces existe una función f computable en tiempo polinomial tal que $x \in \mathcal{L}'$ sii $f(x) \in \mathcal{L}$ para todo $x \in \{0,1\}^*$. Como $\mathcal{L} \in \mathbf{P}$, $\chi_{\mathcal{L}}$ es computable en tiempo polinomial. Entonces tenemos que $x \in \mathcal{L}'$ sii $\chi_{\mathcal{L}}(f(x)) = 1$. Observar que $\chi_{\mathcal{L}} \circ f$ es computable en tiempo polinomial. Luego $\chi_{\mathcal{L}'}$ es computable en tiempo polinomial, y esto quiere decir que $\mathcal{L}' \in \mathbf{P}$. □

[8] En castellano algunos lo llaman **NP**-*difícil* o **NP**-*duro*. Elegí llamarlos con la palabra en inglés porque creo que es la jerga que más se usa acá.

El siguiente resultado muestra que para probar que $\mathbf{P} = \mathbf{NP}$, alcanza con probar que algún problema **NP-completo** está en $\mathbf{P}$.

Teorema 9. *Si $\mathcal{L} \in$ **NP-completo**, entonces $\mathcal{L} \in \mathbf{P}$ sii $\mathbf{P} = \mathbf{NP}$.*

Demostración. Supongamos que $\mathcal{L} \in$ **NP-completo**. Para la implicación $\Rightarrow$, supongamos que $\mathcal{L} \in \mathbf{P}$. Por hipótesis $\mathcal{L} \in$ **NP-completo**. Luego $\mathcal{L} \in$ **NP-completo** $\cap \mathbf{P}$. Entonces $\mathcal{L} \in$ **NP-hard** $\cap \mathbf{P}$. Aplicando el Teorema 8, concluimos que $\mathbf{P} = \mathbf{NP}$. Para la implicación $\Leftarrow$, supongamos que $\mathbf{P} = \mathbf{NP}$. Por hipótesis $\mathcal{L} \in \mathbf{NP}$, de modo que $\mathcal{L} \in \mathbf{P}$. □

Ejemplo 6. Llamemos M_y a la máquina determinística tal que $\langle M \rangle = y$ y definamos el siguiente problema:

Problema: TMSAT

$$\mathsf{TMSAT} = \{\langle y, x, 1^n, 1^t \rangle \mid \exists u \in \{0,1\}^n \ M_y(xu) = 1 \text{ en a lo sumo } t \text{ pasos}\}$$

Proposición 6. TMSAT $\in$ **NP-completo**.

Demostración. Es claro que TMSAT $\in \mathbf{NP}$. Tomemos $\mathcal{L} \in \mathbf{NP}$ y veamos que $\mathcal{L} \leq_{\mathrm{p}}$ TMSAT. Como $\mathcal{L} \in \mathbf{NP}$, existen polinomios p, q y una máquina determinística M tal que M corre en tiempo $q(n)$ y

$$x \in \mathcal{L} \qquad \text{sii} \qquad \exists u \in \{0,1\}^{p(|x|)} \ M(xu) = 1.$$

Definimos $f(x) = \langle \langle M \rangle, x, 1^{p(|x|)}, 1^{q(|x|+p(|x|))} \rangle$. Es claro que f es computable en tiempo polinomial. Así, tenemos que

$$\begin{aligned} f(x) \in \mathsf{TMSAT} \quad &\text{sii} \quad \langle \langle M \rangle, x, 1^{p(|x|)}, 1^{q(|x|+p(|x|))} \rangle \in \mathsf{TMSAT} \\ &\text{sii} \quad \exists u \in \{0,1\}^{p(|x|)} \ M(xu) = 1 \text{ en} \leq q(|xu|) = q(|x| + p(|x|)) \text{ pasos} \\ &\text{sii} \quad x \in \mathcal{L} \end{aligned}$$

y esto prueba que $\mathcal{L} \leq_{\mathrm{p}}$ TMSAT vía f.

□

4.1. Lógica proposicional

En esta sección vamos a definir muy brevemente los ingredientes básicos de la lógica proposicional. Va a ser una herramienta importante para probar un teorema fundamental de la Teoría de la Complejidad, el Teorema de Cook-Levin de §4.4, y se va a volver a usar en §8.

Fijamos un conjunto PROP de **variables proposicionales**. Una **fórmula booleana** se forma con la gramática

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \tag{5}$$

donde $p \in$ PROP. Usaremos $(\varphi_1 \wedge \varphi_2 \wedge \varphi_3)$ para referirnos a $(\varphi_1 \wedge (\varphi_2 \wedge \varphi_3))$ y lo mismo para $\vee$. Muchas veces omitiremos los paréntesis exteriores. El **tamaño** de una fórmula booleana es la cantidad de símbolos que contiene.

Una **valuación** es una función $v : \mathsf{PROP} \to \{0,1\}$. Definimos la noción de verdad de una fórmula booleana para una valuación, notado $v \models \varphi$, por inducción en φ:

- si $p \in \mathsf{PROP}$, entonces $v \models p$ si $v(p) = 1$
- $v \models \neg\varphi$ si no $v \models \varphi$
- $v \models \varphi \wedge \psi$ si $v \models \varphi$ y $v \models \psi$
- $v \models \varphi \vee \psi$ si $v \models \varphi$ o $v \models \psi$

Como $\wedge$ y $\vee$ son asociativos, escribiremos $(\varphi_1 \vee \varphi_2 \vee \varphi_3)$ en lugar de $((\varphi_1 \vee \varphi_2) \vee \varphi_3)$ o de $(\varphi_1 \vee (\varphi_2 \vee \varphi_3))$ y lo mismo para $\wedge$. Decimos que

- φ **es verdadera para** v o que v **satisface** φ si $v \models \varphi$
- φ **es falsa para** v o que v **no satisface** φ si $v \models \varphi$ es falso, notado $v \not\models \varphi$
- φ es **satisfacible** si existe una valuación v tal que $v \models \varphi$
- φ es una **tautología** si $v \models \varphi$ para toda valuación v

Ejemplo 7. Supongamos que $x_1, x_2, x_3 \in \mathsf{PROP}$. La fórmula booleana $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_4)$ es satisfacible por cualquier valuación v tal que $v(x_1) = 1, v(x_2) = 0, v(x_3) = 1, v(x_4) = 1$.

Usaremos la notación $\varphi(p_1, \ldots, p_n)$ para marcar que las variables proposicionales que aparecen en φ están entre $p_1, \ldots, p_n$ (podrían ser solo algunas de ellas, quizá todas, quizá ninguna).

El siguiente resultado lo tenemos incorporado y lo usamos automáticamente: el valor de verdad de una fórmula booleana solo depende del valor de las variables proposicionales que participan de la fórmula.

Proposición 7. *Sea $\varphi(p_1, \ldots, p_n)$ una fórmula booleana. Si v y v' son dos valuaciones tales que $v(p_i) = v'(p_i)$ para $i \in \{1, \ldots, n\}$, entonces $v \models \varphi(p_1, \ldots, p_n)$ sii $v' \models \varphi(p_1, \ldots, p_n)$.*

Entonces, para saber si $v \models \varphi(p_1, \ldots, p_n)$ alcanza con conocer $v \restriction \{p_1, \ldots, p_n\}$. Codificaremos valuaciones parciales con tuplas en binario o, lo que es lo mismo, cadenas en binario. Usaremos $v \in \{0,1\}^n$ para referirnos a la función $\{p_i \mapsto v(i) \mid i = 1 \ldots n\}$.

Ejemplo 8. La valuación parcial $\{p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 1\}$ se representa como $(0,0,1)$ o como 001. Entonces $001 \models \neg p_1 \wedge \neg p_2 \wedge p_3$ y $001 \not\models p_1 \vee p_2$.

En algunas ocasiones, extenderemos la gramática de (5) a

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid 0 \mid 1, \tag{6}$$

donde 0 y 1 representan las constantes para falso y verdadero respectivamente. Más formalmente, extendemos la noción de verdad con

- $v \models 0$ nunca
- $v \models 1$ siempre

Todos los resultados que demos sobre fórmulas booleanas valdrán con o sin constantes. Su uso es meramente por comodidad, dado que 0 es equivalente a $(p \wedge \neg p)$ y 1 es equivalente a $(p \vee \neg p)$.

Una fórmula booleana está en **forma normal conjuntiva (CNF)** si es de la forma

$$\underbrace{(a_{11} \vee \cdots \vee a_{1n_1})}_{\text{cláusula}} \wedge \underbrace{(a_{21} \vee \cdots \vee a_{2n_2})}_{\text{cláusula}} \wedge \cdots \wedge \underbrace{(a_{m1} \vee \cdots \vee a_{mn_m})}_{\text{cláusula}}.$$

Es decir, una fórmula booleana en CNF es una conjunción de cláusulas, donde cada **cláusula** es una disyunción de literales donde a_{ij}, y cada **literal** es una expresión de la forma p o $\neg p$ para algún $p \in \mathsf{PROP}$. Una fórmula booleana está en **3CNF** si está en CNF y cada cláusula tiene a lo sumo 3 literales.

Existen muchas formas de codificar fórmulas booleanas con palabras de $\{0,1\}^*$. Una posibilidad es numerar las variables proposicionales $p_1, p_2, p_3, \cdots \in \mathsf{PROP}$ y definir la siguiente codificación:

$$\begin{aligned} \langle p_i \rangle &= 1^i 0 & \langle \neg\varphi \rangle &= 0010\langle\varphi\rangle \\ \langle 0 \rangle &= 0000 & \langle \varphi \wedge \psi \rangle &= 0011\langle\varphi\rangle\langle\psi\rangle \\ \langle 1 \rangle &= 0001 & \langle \varphi \vee \psi \rangle &= 0100\langle\varphi\rangle\langle\psi\rangle \end{aligned} \tag{7}$$

Problema: Satisfacción booleana en CNF / en 3CNF

$$\mathsf{SAT} = \{\langle\varphi\rangle : \varphi \in \text{CNF es satisfacible}\}$$
$$\mathsf{3SAT} = \{\langle\varphi\rangle : \varphi \in \text{3CNF es satisfacible}\}$$

Teorema 10. $\mathsf{SAT}, \mathsf{3SAT} \in \mathbf{NP}$.

Demostración. Alcanza con verlo para SAT. Definimos una máquina determinística M que recibe como entrada $\langle x, u\rangle$ y hace esto:

si x no representa una fórmula booleana, rechazar.
si no, supongamos que $x = \langle\varphi\rangle$ y supongamos que φ tiene m variables proposicionales
  si $|u| < m$ rechazar
  si no, aceptar si $u \restriction m \models \varphi$ y rechazar en caso contrario

Tenemos que $\langle\varphi\rangle \in \mathsf{SAT}$ sii $\exists u \in \{0,1\}^{|\langle\varphi\rangle|}\ M(\langle x, u\rangle) = 1$. Como M corre en tiempo polinomial, $\mathsf{SAT} \in \mathbf{NP}$. El certificado de M es la valuación que hace verdadera a φ: es una palabra en $\{0,1\}^{p(|\langle\varphi\rangle|)}$, donde $p(n) = n$. Estamos usando que $\langle\varphi\rangle$ tiene tamaño al menos la cantidad de variables proposicionales de φ. □

Ejemplo 9. Sea $\varphi = (\neg p_1 \vee p_2 \vee \neg p_3) \wedge (p_1 \vee p_2 \vee p_3)$. En la demostración del Teorema 10, cualquier $u = 001v$, con $v \in \{0,1\}^{|\langle\varphi\rangle|-3}$ es un certificado para φ porque $001 \models \varphi$.

4.2. Representación de funciones booleanas

Proposición 8. *Para toda $F : \{0,1\}^\ell \to \{0,1\}$ existe una fórmula booleana $\varphi_F(p_1, \ldots, p_\ell)$ en CNF tal que $v \models \varphi_F$ sii $F(v) = 1$ para todo $v \in \{0,1\}^\ell$. Más aún, φ_F se computa en tiempo polinomial a partir de $\langle F\rangle$ y tiene tamaño $O(\ell 2^\ell)$.*

Demostración. Observar que $|\langle F\rangle| = O(2^\ell)$. Se puede ver que

$$\varphi_F = \bigwedge_{v:F(v)=0} \neg \left(\bigwedge_{i:v(i)=1} p_i \wedge \bigwedge_{i:v(i)=0} \neg p_i \right) = \bigwedge_{v:F(v)=0} \underbrace{\left(\bigvee_{i:v(i)=1} \neg p_i \vee \bigvee_{i:v(i)=0} p_i \right)}_{\text{tamaño } O(\ell)},$$

donde los v representan palabras en $\{0,1\}^\ell$. La última expresión tiene tamaño $O(\ell 2^\ell)$. □

Ejemplo 10. Supongamos la función $F : \{0,1\}^3 \to \{0,1\}$ definida de la siguiente manera:

$v(0)$	$v(1)$	$v(2)$	$F(v)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Entonces, la siguiente fórmula booleana φ_F satisface que $v \models \varphi_F$ sii $F(v) = 1$ para todo $v \in \{0,1\}^3$:

$$\varphi = (p_1 \vee p_2 \vee p_3) \wedge (p_1 \vee p_2 \vee \neg p_3) \wedge (\neg p_1 \vee p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee \neg p_3)$$

La fórmula booleana φ_F está en CNF y cada cláusula corresponde a una palabra $v \in \{0,1\}^3$ tal que $F(v) = 0$. Dada una tal palabra $v = v_1 v_2 v_3$ ($v_i \in \{0,1\}$), definimos la cláusula como $(\neg_1)p_1 \vee (\neg_2)p_2 \vee (\neg_3)p_3$, donde $(\neg_i)$ representa a la cadena vacía si $v_i = 0$ y $(\neg_i)$ representa a la negación $\neg$ si $v_i = 1$.

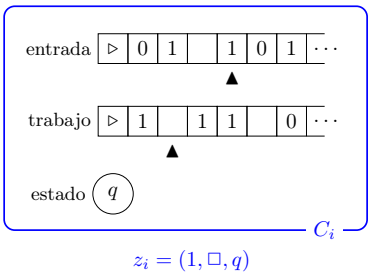


Figura 14: Una mini-configuración de una máquina con una sola cinta de trabajo y sin cinta de salida.

Corolario 1. *Para toda $F : \{0,1\}^\ell \to \{0,1\}^k$ existe una fórmula booleana $\varphi_F(p_1, \ldots, p_{\ell+k})$ en CNF tal que $uv \models \varphi_F$ sii $F(u) = v$ para todo $u \in \{0,1\}^\ell$, $v \in \{0,1\}^k$. Más aún, φ_F se computa en tiempo polinomial a partir de $\langle F \rangle$ y tiene tamaño $O((\ell + k)2^{\ell+k})$.*

Demostración. Considerar $G : \{0,1\}^{\ell+k} \to \{0,1\}$ definida como $G(uv) = 1$ sii $F(u) = v$ y aplicar la Proposición 8. □

4.3. Mini-configuraciones

Para simplificar, consideremos una máquina determinística $M = (\Sigma, Q, \delta)$ sin cinta de salida, con una cinta de entrada y con una sola cinta de trabajo (que, como toda máquina sin cinta de salida, funciona como salida también). Todo lo que digamos a continuación se puede generalizar a máquinas con cualquier cantidad de cintas de trabajo. Supongamos un cómputo

$$C_0, \ldots, C_\ell \tag{8}$$

de M con entrada x. La **mini-configuración en el paso** i es una tupla $z_i = (a_i, b_i, c_i) \in \Sigma \times \Sigma \times Q$ tal que

- a_i es el símbolo leído por la cabeza de entrada en C_i
- b_i es el símbolo leído en la cinta de trabajo en C_i
- c_i es el estado de C_i

La Figura 14 muestra un ejemplo de mini-configuración C_i. Observar que la mini-configuración z_i no tiene toda la información de C_i sino una parte. Es una especie de resumen de C_i. Veremos que bajo algunas hipótesis la información de z_i es suficiente para los razonamientos que necesitamos.

Supongamos que, además, M es oblivious. Recordemos que esto quiere decir que podemos calcular la posición de las cabezas de entrada y trabajo en el cómputo de M con entrada x en función de $|x|$ dado en el cómputo (8) y el número de paso (pero independiente de x).

Se puede ver que las siguientes tres funciones son computables en tiempo polinomial:

- $e(i, n)$ = posición de la cabeza de la cinta de entrada en el paso i en el cómputo de M con entrada 0^n
- $t(i, n)$ = posición de la cabeza de la cinta de trabajo en el paso i en el cómputo de M con entrada 0^n
- $prev(i, n) = \max\{j < i \mid t(j, n) = t(i, n)\} \cup \{1\}$, es decir $prev(i, n)$ es el máximo paso $j < i$ en el cómputo de M con entrada 0^n tal que la posición de la cabeza de trabajo en el paso j coincide con la posición en el paso i; o 1 si no existe tal j.[9]

[9] Numeramos las posiciones de las cintas desde 0, de modo que inicialmente la posición de todas las cabezas es 1 (ver Figura 4-izquierda).

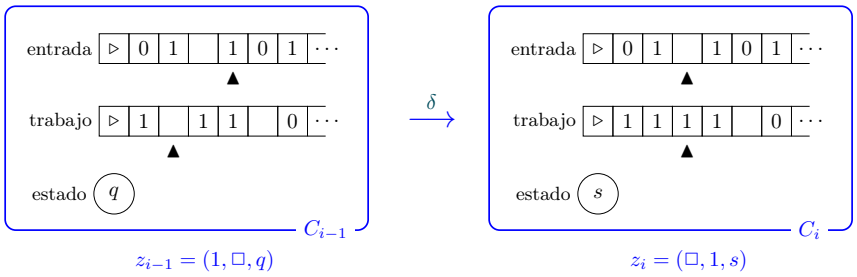


Figura 15: La evolución en un paso de una mini-configuración a otra.

Vamos a considerar ahora la evolución en un paso de C_{i-1} a C_i en el cómputo (8), pero analizando solamente las mini-configuraciones z_{i-1} y z_i (ver Figura 15).

Recordemos que z_i es la i-ésima mini-configuración en el cómputo de M con entrada x. Entonces $z_0 = (x(0), \square, q_0)$ y para $i > 0$ calculamos z_i con:

- el estado y los símbolos sobre los que se sitúan las cabezas en el paso $i-1$; esta información está en z_{i-1}
- la función de transición δ de M
- el contenido de la cinta de entrada (recordar que es de solo lectura y por lo tanto no cambia a lo largo del cómputo) en la posición $e(i, |x|)$
- el contenido de la cinta de trabajo en la posición $prev(i, |x|)$; esta información está en $z_{prev(i,|x|)}$. Esto es necesario porque la cabeza la cinta de trabajo se mueve. Evidentemente la información de qué símbolo está siendo leído en el i-ésimo paso del cómputo está en C_i. Sin embargo, z_i codifica solo una parte de la información en C_i; en particular, no codifica todo el contenido de la cinta de trabajo sino solamente el símbolo que está siendo leído por la cabeza en el paso i del cómputo en cuestión, es decir, el símbolo en la posición $t(i, |x|)$. Entonces, tenemos que buscar la información del símbolo leído por la cabeza de la cinta de trabajo en z_j, donde j es el máximo número menor que i tal que a posición de la cabeza de la cinta de trabajo en z_j fue justamente $t(i, |x|)$, o 1 (que representa la segunda posición de la cinta de trabajo) si no existe tal j. El hecho de que sea el máximo es importante, porque quiere decir que no fue modificado con posterioridad antes del paso i. El símbolo que tenía escrito la cinta de trabajo en la posición en el paso j sobrevive hasta el paso i sin modificaciones intermedias. Este j que buscamos es justamente $prev(i, |x|)$.

Considerando siempre la máquina M y el cómputo de (8), podemos codificar cada mini-configuración $z \in \Sigma \times \Sigma \times Q$ con $\langle z \rangle \in \{0,1\}^k$ con $k = 4 + \lceil \log |Q| \rceil$ de esta manera:

- $00, 11, 01, 10$ codifica cada símbolo de $\Sigma = \{0, 1, \triangleright, \square\}$
- cada estado de Q se codifica con una cadena en $\{0,1\}^{\lceil \log |Q| \rceil}$, fijando que $0 \ldots 0$ codifica q_0 y que $0 \ldots 1$ codifica q_f

Notemos que k depende solo de M. Para $i > 0$, definimos $F : \{0,1\}^k \times \{0,1\}^k \times \{0,1\}^2 \to \{0,1\}^k$ como

$$F\Big(\underbrace{\langle z_{i-1} \rangle}_{\substack{\text{mini-}\\ \text{configuración}\\ \text{anterior}\\ (k \text{ variables})}} , \underbrace{\langle z_{prev(i,|y|)} \rangle}_{\substack{\text{información de}\\ \text{la cinta de tra-}\\ \text{bajo}\\ (k \text{ variables})}} , \underbrace{\langle x(e(i,|y|)) \rangle}_{\substack{\text{codificación del}\\ \text{bit actual leído}\\ \text{en la cinta de}\\ \text{entrada}\\ (2 \text{ variables})}} \Big) = \langle z_i \rangle. \tag{9}$$

y como $F(x) = 0^k$ para las entradas que falten definir (la ecuación (9) solo define parcialmente a F; este último caso es solo para garantizar que F sea función pero no es importante). La

función booleana F representa la evolución en un paso del cómputo (8) de M con entrada y, pero representando la información en cada paso con una mini-configuración, que a su vez se codifica con k bits cada una. Por el Corolario 1, existe una fórmula booleana

$$\varphi_F(\bar{p}, \bar{q}, \bar{r}, \bar{s})$$

con variables libres

- $\bar{p} = p_1, \ldots, p_k$, que codifica $\langle z_{i-1} \rangle$
- $\bar{q} = q_1, \ldots, q_k$, que codifica $\langle z_{prev(i,|y|)} \rangle$
- $\bar{r} = r_1, r_2$, que codifica $\langle y(e(i, |y|)) \rangle$
- $\bar{s} = s_1, \ldots, s_k$, que codifica $\langle z_i \rangle$

en CNF tal que para todo $\bar{a}, \bar{b}, \bar{d} \in \{0,1\}^k$ y $\bar{c} \in \{0,1\}^2$:

$$\bar{a}\bar{b}\bar{c}\bar{d} \models \varphi_F(\bar{p}, \bar{q}, \bar{r}, \bar{s}) \qquad \text{sii} \qquad \bar{d} = F(\bar{a}, \bar{b}, \bar{c}).$$

Más aun, podemos computar φ_F a partir de $\langle F \rangle$ en tiempo polinomial y φ_F tiene tamaño a lo sumo $(3k+2)2^{3k+2}$. Observar que, como k es constante, el tamaño de φ_F también es constante.

4.4. Teorema de Cook-Levin

En esta sección probaremos el siguiente resultado fundamental.

Teorema 11. SAT $\in$ **NP-hard**.

Veamos primero un argumento tentador pero incorrecto. Fijemos $\mathcal{L} \in \mathbf{NP}$ y veamos que $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$. Como $\mathcal{L} \in \mathbf{NP}$, existe una máquina determinística M tal que M corre en tiempo $t(n)$, con t un polinomio, y existe un polinomio p tal que para todo $x \in \{0,1\}^*$ tenemos que $x \in \mathcal{L}$ sii $\exists u \in \{0,1\}^{p(|x|)}\ M(xu) = 1$. Definimos $F_x : \{0,1\}^{p(|x|)} \to \{0,1\}$ como $F_x(u) = M(xu)$. Por el Corolario 1, podemos computar[10] $\varphi_x(q_1, \ldots, q_{p(|x|)}) \in$ CNF tal que

$$u \models \varphi_x \qquad \text{sii} \qquad F_x(u) = 1 \qquad \text{sii} \qquad M(xu) = 1.$$

Luego $x \in \mathcal{L}$ sii φ_x es satisfacible sii $\varphi_x \in \mathsf{SAT}$. El problema es que φ_x tiene tamaño exponencial: por el Corolario 1 tiene tamaño $O(p(|x|)2^{p(|x|)})$. Entonces, no podemos afirmar que $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$ porque si bien la función $x \mapsto \varphi_x$ es computable, no es computable en tiempo polinomial.

Como veremos a continuación, el argumento es más sofisticado y necesita trabajar con las mini-configuraciones.

Demostración del Teorema 11. Fijemos $\mathcal{L} \in \mathbf{NP}$ y veamos que $\mathcal{L} \leq_{\mathrm{p}} \mathsf{SAT}$. Como $\mathcal{L} \in \mathbf{NP}$, existe una máquina determinística M tal que M corre en tiempo $t(n)$, con t un polinomio, y existe un polinomio p tal que

$$x \in \mathcal{L} \qquad \text{sii} \qquad \exists u \in \{0,1\}^{p(|x|)}\ M(xu) = 1. \tag{10}$$

Por la Proposición 3, podemos suponer que M es oblivious y por la Proposición 5, podemos suponerla también sin cinta de salida. Dado $x \in \{0,1\}^*$ construiremos una fórmula booleana $\varphi_x \in$ CNF en tiempo polinomial tal que

$$x \in \mathcal{L} \qquad \text{sii} \qquad \varphi_x \in \mathsf{SAT}. \tag{11}$$

[10] El argumento del Corolario 1 se basa en el de la Proposición 8. Este último argumento es *uniforme* en el sentido de que existe una máquina que dada una representación para F, computa una representación para φ_F como en el enunciado de la Proposición 8.

Es importante marcar que de ahora en más la máquina M está fija, pero x es variable. Tenemos que

$$\begin{aligned} x \in \mathcal{L} \quad & \text{sii} \quad \exists u \in \{0,1\}^{p(|x|)}\ M(xu) = 1 \\ & \text{sii} \quad \text{existe } u \in \{0,1\}^{p(|x|)} \text{ y existe un cómputo } C_0, \ldots, C_{t(|xu|)} \text{ a partir} \\ & \qquad \text{de la entrada } xu \text{ tal que la salida en } C_{t(|xu|)} \text{ es } 1 \end{aligned} \tag{12}$$

y esto sucede sii existe una codificación de la entrada $y \in \{0,1\}^{2n+4}$ con $n = |x| + p(|x|)$ y una secuencia de mini-configuraciones $z_0, \ldots, z_m$, $z_i \in \{0,1\}^k$ (k depende solo de M, y M está fija), con $m = t(n) = t(|x| + p(|x|))$, tal que:

1. la cinta de entrada (codificada con y) empieza con x (parámetro) y sigue con $u \in \{0,1\}^*$
2. z_0 es la mini-configuración que corresponde a la configuración inicial de M a partir de y
3. z_i evoluciona en un paso en z_{i+1}
4. z_m es una mini-configuración final de M aceptadora (es decir, el estado en z_m es q_f y el símbolo en la cinta de trabajo es 1)

Cada una de estas cuatro condiciones se puede expresar con una fórmula booleana. Entonces, para cada $i = 1 \ldots 4$, definiremos una fórmula booleana

$$\psi_i(\underbrace{p_0, \ldots, p_{2n+3}}_{\text{entrada } y}, \underbrace{q_1^0, \ldots q_k^0}_{z_0}, \ldots, \underbrace{q_1^m, \ldots, q_k^m}_{z_m}) \tag{13}$$

con $2n + 4 + k(m+1) = O(t(|x| + p(|x|)))$ variables tal que *a)* expresa el ítem i de arriba, *b)* está en CNF y *c)* es computable en tiempo polinomial a partir de x. Finalmente, definimos

$$\varphi_x = \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4,$$

computable en tiempo polinomial[11] a partir de x. Retomando (12), tenemos que $\varphi_x \in \mathsf{SAT}$ sii $x \in \mathcal{L}$.

Lo que nos falta para terminar la prueba es expresar con una fórmula booleana ψ_i para cada una de las cuatro condiciones $i = 1 \ldots 4$. Recordemos que la entrada de M es xu, y que $n = |xu| = |x| + p(|x|)$.

1. La cinta de entrada (codificada con y) empieza con x (parámetro) y sigue con $u \in \{0,1\}^*$. Recordemos que $00, 11, 01, 10$ codifica cada símbolo de $\Sigma = \{0, 1, \triangleright, \square\}$. Entonces ψ_1 va a expresar que la entrada y representa "$\triangleright\ x\ u\ \square$", y esto se logra mediante:

- $y(0)y(1) = 01$ (marca $\triangleright$)
- $y(2j+2)$ y $y(2j+3)$ codifican a $x(j)$ para $0 \leq j \leq |x| - 1$
- $y(2j)$ y $y(2j+1)$ tienen el mismo valor para $|x| + 1 \leq j \leq n$
- $y(2n+2)y(2n+3) = 10$ (marca $\square$)

En concreto,

$$\begin{aligned} \psi_1 = & \neg p_0 \wedge p_1 && \text{codificación de } \triangleright \\ & \wedge \bigwedge_{j=0\ldots|x|-1} \begin{cases} p_{2j+2} \wedge p_{2j+3} & \text{si } x(j) = 1 \\ \neg p_{2j+2} \wedge \neg p_{2j+3} & \text{si } x(j) = 0 \end{cases} && \text{codificación de entrada } x \\ & \wedge \bigwedge_{i=|x|+1\ldots n} p_{2i} \leftrightarrow p_{2i+1} && \text{certificado } u \in \{0,1\}^* \\ & \wedge p_{2n+2} \wedge \neg p_{2n+3} && \text{codificación de } \square \end{aligned}$$

[11] Se puede verificar que $|\varphi_x| = O(|x| + t(|x| + p(|x|)))$

El símbolo $\leftrightarrow$ es la doble implicación, que se puede escribir en CNF como $p \leftrightarrow q = (p \to q) \wedge (q \to p) = (\neg p \vee q) \wedge (p \vee \neg q)$. Notemos que no se especifican los valores de las variables booleanas $p_{2|x|+2}, \ldots, p_{2n+1}$, correspondientes a (la codificación de) u sino que solamente se especifica que u está formada por símbolos de $\{0,1\}$. Este grado de libertad es crucial en la demostración, porque es el enlace con la satisfacibilidad de la fórmula booleana que vamos a construir Observemos también que $|\psi_1| = O(n)$.

2. z_0 es la configuración inicial. ψ_2 va a expresar que $z_0 = (x(0), \square, q_0)$ y se expresa con

$$\begin{aligned}
\psi_2 = &\begin{cases} q_1^0 \wedge q_2^0 & \text{si } x(0) = 1 \\ \neg q_1^0 \wedge \neg q_2^0 & \text{si } x(0) = 0 \end{cases} && \text{codificación de } x(0) \\
&\wedge q_3^0 \wedge \neg q_4^0 && \text{codificación de } \square \\
&\wedge \bigwedge_{i=5\ldots k} \neg p_i^0 && \text{codificación de } q_0
\end{aligned}$$

Observemos que $|\psi_1| = O(1)$ (una vez más, k depende solo de M pero M está fija, de modo que k es constante).

3. z_i evoluciona en un paso en z_{i+1}. Recordemos que y es la codificación de xu y que $n = |xu| = |x| + p(|x|)$. Para $0 < i \leq m$ la condición

$$\underbrace{\langle z_i \rangle}_{k} = F\left(\underbrace{\langle z_{i-1} \rangle}_{k}, \underbrace{\langle z_{prev(i,n)} \rangle}_{k}, \underbrace{\langle xu(e(i,n)) \rangle}_{2}\right)$$

se expresa con una fórmula booleana $\psi_3^i \in$ CNF, con ayuda la fórmula booleana φ_F del Corolario 1. En concreto

$$\psi_3^i = \varphi_F\left(\underbrace{q_1^{i-1}, \ldots, q_k^{i-1}}_{\langle z_{i-1} \rangle}, \underbrace{q_1^{prev(i,n)}, \ldots, q_k^{prev(i,n)}}_{\langle z_{prev(i,n)} \rangle}, \underbrace{p_{2e(i,n)}, p_{2e(i,n)+1}}_{\langle xu(e(i,n)) \rangle}, \underbrace{q_1^i, \ldots, q_k^i}_{\langle z_i \rangle}\right) \tag{14}$$

es tal que para todo $\bar{a}, \bar{b}, \bar{d} \in \{0,1\}^k$ y $\bar{c} \in \{0,1\}^2$, tenemos

$$\bar{a}\bar{b}\bar{c}\bar{d} \models \varphi_F(\bar{p}, \bar{q}, \bar{r}, \bar{s}) \qquad \text{sii} \qquad \bar{d} = F(\bar{a}, \bar{b}, \bar{c}).$$

La primera celda de la cinta de entrada tiene posición 0. Podemos suponer que $e(i,n) \leq |xu| + 1$ porque M con entrada $y = xu$ no necesita leer más allá del primer blanco después de y. Computamos ψ_3^i en tiempo polinomial y tiene tamaño $O(k2^{3k}) = O(1)$ porque k es constante. Finalmente construimos $\psi_3 = \bigwedge_{i=1,\ldots,m} \psi_3^i$ en tiempo polinomial y expresa

$$(\forall i = 1, \ldots, t(n))\ z_i = F(z_{i-1}, z_{prev(i,n)}, \langle xu(e(i,n)) \rangle).$$

Observemos que $|\psi_3| = O(k2^{3k}m) = O(m) = O(t(n)) = O(t(|x| + p(|x|)))$ porque k es constante y $m = t(n) = t(|x| + p(|x|)$.

4. z_m es una configuración final de M aceptadora. La propiedad que dice que z_m es de la forma $(*, 1, q_f)$, se expresa con ψ_4 con $|\psi_4| = O(1)$ análogo a ψ_2:

$$\begin{aligned}
\psi_4 = &\ q_3^m \wedge q_4^m && \text{codificación de la salida} \\
&\wedge p_k^m \wedge \bigwedge_{i=5\ldots k-1} \neg p_i^m && \text{codificación de } q_f
\end{aligned}$$

Esto concluye la demostración. $\square$

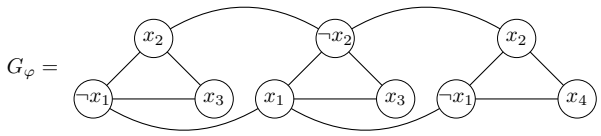


Figura 16: El grafo G_φ, donde $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$.

Ejercicio 2. Probar que $\mathsf{SAT} \leq_\mathrm{p} \mathsf{3SAT}$.

Como consecuencia del Teorema 10, Teorema 11 y el Ejercicio 2 obtenemos

Corolario 2. $\mathsf{SAT}, \mathsf{3SAT} \in$ **NP-completos**.

4.5. Ejemplos de problemas NP-completos

En esta sección mencionaremos algunos problemas **NP-completos**. Daremos la demostración solo del primero.

Proposición 9. $\mathsf{INDSET} \in$ **NP-completo**.

Demostración. Ya vimos que INDSET es **NP**. Para ver que INDSET es **NP-hard**, probamos que $\mathsf{3SAT} \leq_\mathrm{p} \mathsf{INDSET}$. Consideremos una fórmula booleana

$$\varphi = (l_{11} \vee l_{12} \vee l_{13}) \wedge (l_{21} \vee l_{22} \vee l_{33}) \wedge \cdots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$$

en 3CNF con m cláusulas (l_{ij} son literales) y variables $x_1, \ldots, x_n$.

Definimos un grafo G_φ con $3m$ vértices. Cada vértice corresponde a cada variable de cada cláusula. Supongamos que z es un vértice de G_φ correspondiente a l_{ij} y que z' es un vértice de G_φ correspondiente a $l_{i'j'}$. Definimos una arista entre z y z' si $i = i'$ o l_{ij} es la negación de $l_{i'j'}$ (o viceversa). Ver Figura 16. Es claro que (la codificación de) G_φ es computable en tiempo polinomial en $|\varphi|$.

Probamos que $\langle\varphi\rangle \in \mathsf{SAT}$ sii $\langle G_\varphi, m\rangle \in \mathsf{INDSET}$, es decir que φ es satisfacible sii G_φ tiene un conjunto independiente de al menos m vértices.

- $\Rightarrow$ Supongamos que $v \models \varphi$. Entonces para todo $i = 1, \ldots, m$ tenemos $v \models l_{i1} \vee l_{i2} \vee l_{i3}$, de modo que $v \models l_{ij}$ para algún j. Sea S el conjunto de vértices $z_1, \ldots, z_m$ tal que z_i corresponde a l_{ij} y $v \models l_{ij}$. S es independiente: si existiera una arista entre z_i y $z_{i'}$ sería porque 1) $i = i'$ o 2) z_i corresponde a l_{ij}, $z_{i'}$ corresponde a $l_{i'j'}$ y l_{ij} es la negación de $l_{i'j'}$ (o viceversa). Es fácil ver que ninguna puede pasar.
- $\Leftarrow$ Si S es un conjunto independiente en G_φ de m vértices, tenemos exactamente un vértice en cada "triángulo". Definimos la valuación v de esta manera: para cada $z \in S$, si z corresponde a x_i definimos $v(x_i) = 1$ y si z corresponde a $\neg x_i$ definimos $v(x_i) = 0$. Para todas las otras variables x para las que no está definido v, definimos $v(x)$ de forma arbitraria. Notemos que v está bien definida porque S es independiente y que $v \models \varphi$.

Esto termina la demostración. □

Un **camino hamiltoniano** en un grafo dirigido G es un camino que visita todos los vértices de G_φ exactamente una vez.

Problema: Camino hamiltoniano

$$\mathsf{CAMHAM} = \{\langle G\rangle : G \text{ tiene un camino hamiltoniano}\}$$

Proposición 10. $\mathsf{CAMHAM} \in$ **NP-completo**.

Dadas n ciudades, representamos la distancia entre cada par de ciudades por medio de una matriz M de $n \times n$.

Problema: Problema del viajante de comercio o *Travelling Salesman Problem*

$$\mathsf{TSP} = \{\langle M, k\rangle : \begin{array}{l}\text{hay una ruta de distancia a lo sumo } k \text{ (de acuerdo a } M\text{) que visita todas}\\ \text{las ciudades de } G \text{ exactamente una vez y al finalizar vuelve a la ciudad}\\ \text{de origen}\end{array}\}$$

Teorema 12. TSP $\in$ **NP-completo**.

Representamos una lista de n ítems en una mochila con su valor y su peso por medio de una lista

$$M = [(v_1, p_1), (v_2, p_2), \ldots, (v_n, p_n)].$$

Problema: Problema de la mochila o *knapsack problem*

$$\mathsf{KNAPSACK} = \{\langle M, v, p\rangle : \begin{array}{l}\text{existe un conjunto de ítems por un valor total de al}\\ \text{menos } v \text{ y con un peso a lo sumo } p\end{array}\}$$

Proposición 11. KNAPSACK $\in$ **NP-completo**.

4.6. La clase coNP

La noción de **NP-hard** y **NP-completo** se aplica a otras clases de complejidad. En general, si **C** es una clase de complejidad, entonces

Clase de complejidad: C-hard, C-completo

$\mathcal{L}$ es **C-hard** si $\mathcal{L}' \leq_{\mathrm{p}} \mathcal{L}$ para todo $\mathcal{L}' \in \mathbf{C}$.
$\mathcal{L}$ es **C-completo** si $\mathcal{L} \in \mathbf{C}$ y $\mathcal{L}$ es **C-hard**.

En realidad, estas son nociones de completitud y hardness[12] para la reducción $\leq_{\mathrm{p}}$. Más adelante, en §6.4, veremos clases de complejidad para las que no tiene sentido usar $\leq_{\mathrm{p}}$ y necesitaremos trabajar con reducciones más débiles.

Usaremos la notación $\overline{\mathcal{L}}$ para referirnos al complemento de $\mathcal{L}$, es decir $\overline{\mathcal{L}} = \{0,1\}^* \setminus \mathcal{L}$.

Clase de complejidad: coC

Si **C** es una clase de complejidad, definimos

$$\mathbf{coC} = \{\mathcal{L} : \overline{\mathcal{L}} \in \mathbf{C}\}.$$

Entonces la clase **coNP** tiene la siguiente definición:

Clase de complejidad: coNP

coNP $= \{\mathcal{L} : \overline{\mathcal{L}} \in \mathbf{NP}\}$. Es decir, **coNP** es la clase de lenguajes $\mathcal{L} \subseteq \{0,1\}^*$ tal que existe un polinomio $p : \mathbb{N} \to \mathbb{N}$ y una máquina determinística $M = (Q, \Sigma, \delta)$ tal que M corre en tiempo polinomial y para todo x:

$$x \in \mathcal{L} \quad \text{sii} \quad \text{para todo } u \in \{0,1\}^{p(|x|)},\ M(\langle x, u\rangle) = 1.$$

Ejercicio 3. Probar que $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$.

Ejercicio 4. Probar que si $\mathbf{P} = \mathbf{NP}$ entonces $\mathbf{NP} = \mathbf{coNP}$.

[12] Así como *completitud* (algunos le dicen *completud*) es la cualidad de ser completo, en inglés *hardness* es la cualidad de ser *hard*. Algunos usan *difícil* en vez de *hard*, y en ese caso, hablaríamos de *dificultad*.

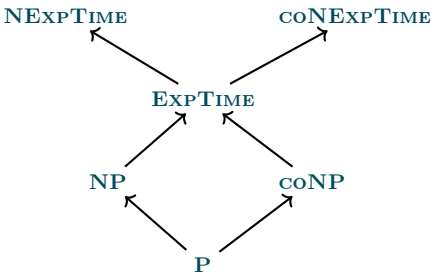


Figura 17: Relación entre las clases **P**, **NP**, **coNP**, **ExpTime**, **NExpTime**y **coNExpTime**. Las flechas representan las inclusiones.

Problema: Tautología

$$\mathsf{TAUT} = \{\langle\varphi\rangle : \varphi \in \mathrm{CNF} \text{ es una tautología}\}$$

Observar que φ es una tautología sii $\neg\varphi$ es insatisfacible, es decir, $\langle\varphi\rangle \in \mathsf{TAUT}$ sii $\langle\neg\varphi\rangle \notin \mathsf{SAT}$.

Ejercicio 5. Probar que TAUT es **coNP-completo**.

4.7. Las clases ExpTime y NExpTime

Presentamos dos nuevas clases de complejidad:

Clase de complejidad: ExpTime, NExpTime

$$\mathbf{ExpTime} = \bigcup_{c>0} \mathbf{DTime}(2^{n^c}).$$
$$\mathbf{NExpTime} = \bigcup_{c>0} \mathbf{NDTime}(2^{n^c}).$$

Son los análogos de **P** y **NP** pero con tiempo exponencial.

Ejercicio 6. Probar que $\mathbf{NP} \subseteq \mathbf{ExpTime}$.

La relación entre las clases que vimos hasta ahora es la siguiente:

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{ExpTime} \subseteq \mathbf{NExpTime}.$$

Más aun, la relación en las clases **coNP**y **coNP**es como se muestra en la Figura 17.

Teorema 13. *Si* $\mathbf{P} = \mathbf{NP}$ *entonces* $\mathbf{ExpTime} = \mathbf{NExpTime}$.

Demostración. Supongamos que $\mathbf{P} = \mathbf{NP}$. Tomemos $\mathcal{L} \in \mathbf{NExpTime}$ y veamos que $\mathcal{L} \in \mathbf{ExpTime}$). Sea N una máquina no-determinística que decide $\mathcal{L}$ en tiempo $c \cdot 2^{n^c}$. Consideremos $\mathcal{L}_{\text{pad}} = \{\langle x, 1^{2^{|x|^c}}\rangle : x \in \mathcal{L}\}$. Veamos que $\mathcal{L}_{\text{pad}} \in \mathbf{NP}$. Definimos una máquina no-determinística N' tal que con entrada $y \in \{0,1\}^*$ hace esto:

si no existe z tal que $y = \langle z, 1^{2^{|z|^c}}\rangle$, rechazar
si no, y es de la forma $\langle z, 1^{2^{|z|^c}}\rangle$
  simular N con entrada z por $c \cdot 2^{|z|^c}$ pasos
  devolver la salida de esta simulación

Es claro que N' corre en tiempo polinomial en $|y|$ y por lo tanto $\mathcal{L}_{\text{pad}} \in \mathbf{NP} = \mathbf{P}$. Veamos que $\mathcal{L}_{\text{pad}} \in \mathbf{DTime}(2^{n^c})$. Definimos una máquina determinística M tal que dada la entrada $x \in \{0,1\}^*$ hace esto:

computa $y = \langle x, 1^{2^{|x|^c}} \rangle$ *(tiempo* $O(2^{|x|^c})$*)*
decide si $y \in \mathcal{L}_{\text{pad}}$ *(sii* $x \in \mathcal{L}$*)* *(tiempo polinomial en* $|y| = O(2^{|x|^{c+d}})$*)*

M corre en tiempo $O(2^{|x|^c})$ y decide $\mathcal{L}$, luego $\mathcal{L} \in \mathbf{DTime}(2^{|x|^{c+d}})$. □

4.8. P vs NP

Desde el punto de vista del modelo de cómputo, **NP** considera cómputos no-determinísticos. A veces se dice que como la máquina es no-determinística entonces "elige" o "adivina" o "inventa" un cómputo aceptador. En realidad la máquina no elige ni adivina ni inventa nada. Esas expresiones informales solo hacen referencia a que para las instancias positivas del problema el programa de la máquina garantiza un cómputo aceptador, y para las instancias negativas garantiza que no habrá ningún cómputo aceptador.

La definición alternativa pero equivalente usa existencia de certificados de tamaño polinomial. En esta definición se ve más clara la diferencia entre encontrar la solución y verificar que una solución sea correcta. Es intuitivo que reconocer si una solución es correcta es más fácil que generarla. Pensemos por ejemplo en las demostraciones matemáticas o en los rompecabezas.

Si **P** es igual o distinto a **NP** es una pregunta abierta importante en computación. No se conoce la respuesta, pero la mayoría apuesta a que son distintos. Eso quiere decir que el no-determinismo de las máquinas o que la información externa de los certificados de tamaño polinomial ayuda, de una manera fundamental, a resolver los problemas en tiempo polinomial.

Por ahora, lo mejor que sabemos hacer con un problema **NP-completo** con una máquina determinística es probar con fuerza bruta todas las combinaciones de certificados o, lo que es lo mismo, todos los posibles cómputos de la máquina no-determinística que resuelve el problema.

5. Separación de clases de complejidad

De las clases de complejidad que estudiamos hasta ahora (ver Figura 17, por ejemplo) conocemos solo algunas inclusiones entre ellas, pero no probamos que ninguna inclusión sea estricta. Justamente, las preguntas sobre las inclusiones estrictas son las que no se conocen en muchas clases de complejidad. Sin embargo, unas pocas sí se conocen. En esta sección mostraremos que existen jerarquías estrictas de tiempos determinísticos y no-determinísticos. En particular, concluiremos que $\mathbf{P} \subsetneq \mathbf{ExpTime}$.

Recordemos la Definición 6 de O grande. Ahora estudiaremos otra noción que formaliza cuándo una función crece más que otra en el límite.

Definición 17. Sean $f, g : \mathbb{N} \to \mathbb{N}$. Decimos que $\boldsymbol{f = o(g)}$ (se lee 'f es o chica de g') si para todo $\epsilon > 0$, tenemos que $f(n) \leq \epsilon \cdot g(n)$ para todo n suficientemente grande o, equivalentemente, si $\lim_n f(n)/g(n) = 0$.

Esta noción es más fuerte de la que de O grande, es decir, si $f = o(g)$ entonces $f = O(g)$.

Ejemplo 11. $4(n+2) \neq o(n)$ pero observar que $4(n+2) = O(n)$; $4(n+2) = o(n^2)$; $n^c = o(2^n)$ para todo $c \in \mathbb{N}$; $k^n = o(2^{n^2})$ para todo $k \in \mathbb{N}$.

En §2.4 codificamos cualquier máquina M con $i = \langle M \rangle \in \{0,1\}^*$. Aquí, M_i representaba la única máquina M con índice i, es decir $\langle M \rangle = i$ y definíamos la máquina universal U que recibía como parámetros $\langle i, x \rangle$ y simulaba $M_i(x)$.

Vamos a introducir una pequeña modificación a la codificación, que no cambia en nada todo lo que ya dijimos. Lo hacemos solo porque simplifica algunos argumentos. Ahora vamos a codificar cada máquina M con *infinitas* cadenas $\langle \langle M \rangle, z \rangle$ para cualquier $z \in \{0,1\}^*$. Cualquier $\langle \langle M \rangle, z \rangle$ representa M. Además, vamos a usar una máquina universal U que utiliza la *nueva* codificación de máquinas: $U(\langle w, x \rangle)$ simula $M_w(x)$ como antes, solo que ahora w es un índice de la *nueva* codificación. Pensemos en términos de programas en Python. Supongamos que P es un programa y definamos P_z como P seguido de z espacios en blanco. Es claro que P_z también es un programa,

que P y P_z computan la misma función y, más aun, que representan el mismo algoritmo (puesto que el intérprete de Python pasa por alto los espacios en blanco del final). Entonces podemos pensar que tenemos infinitas representaciones para un programa P.

Lo importante es que para la nueva codificación $(M_i)_{i\in\{0,1\}^*}$ vale que dada una máquina M existen infinitos i tal que $M = M_i$. Esta es la propiedad que viene bien tener a mano para algunas demostraciones. El Teorema 2 sigue valiendo para esta nueva codificación de máquinas.

5.1. La jerarquía de tiempos determinísticos

El siguiente resultado se prueba con la técnica de diagonalización que vimos en §2.6.

Teorema 14. *Si f, g son construibles en tiempo y cumplen que $f(n)\cdot\log f(n) = o(g(n))$ entonces* $\mathbf{DTime}(f(n)) \subsetneq \mathbf{DTime}(g(n))$.

Demostración. Definimos una máquina determinística D que con entrada x hace esto:

> *calcular $n = |x|$ y $t = g(n)$*
> *simular $U(\langle x, x\rangle)$ por t pasos*
> *si $U(\langle x, x\rangle)$ no terminó en a lo sumo t pasos, devolver 0*
> *si no, devolver $1 - U(\langle x, x\rangle)$*

Es claro que D corre en tiempo $O(g(n))$, de modo que $\mathcal{L}(D) \in \mathbf{DTime}(g(n))$. Supongamos por el absurdo que $\mathcal{L}(D) \in \mathbf{DTime}(f(n))$. Sea M una máquina determinística que decide $\mathcal{L}(D)$, es decir, tal que $\mathcal{L}(M) = \mathcal{L}(D)$ y tal que existe n_0 tal que si $x \in \{0,1\}^*$ y si $|x| \geq n_0$, entonces M con entrada x termina en a lo sumo $c\cdot f(|x|)$ pasos. Como $U(\langle x, x\rangle)$ simula a $M_x(x)$, por el Teorema 2, existe una constante c' tal que $U(\langle x, x\rangle)$ necesita a lo sumo $c'\cdot c\cdot f(|x|)\cdot\log(c\cdot f(|x|))$ pasos para terminar. Existen constantes d y $n_1 \geq n_0$ tal que para todo $n \geq n_1$ tenemos

$$c'\cdot c\cdot f(n)\cdot\log(c\cdot f(n)) \leq d\cdot f(n)\log f(n) \leq g(n).$$

Tomemos x tal que $|x| \geq n_1$ y $M_x = M$. La existencia de este x está garantizada por la nueva codificación de máquinas que elegimos. Tenemos que $U(\langle x, x\rangle)$ termina en a lo sumo $g(|x|)$ pasos. Luego $D(x) = 1 - M_x(x) = 1 - M(x)$. Pero esto es un absurdo porque $\mathcal{L}(D) = \mathcal{L}(M)$. □

Ejemplo 12. $\mathbf{DTime}(n^c) \subsetneq \mathbf{DTime}(n^d)$ si $c < d$.

Notemos que el argumento del Teorema 14 es parecido a lo que hicimos para demostrar que *halt* no es computable en el Teorema 1. Definimos una máquina D tal que D corre en tiempo $O(g(n))$ y D se porta distinto a todas las máquinas que corren en tiempo $O(f(n)\cdot\log f(n))$, porque $D(x)$ 'niega' la salida de $M_x(x)$: cuando $M_x(x) = 1$, $D(x) = 0$ y cuando $M_x(x) = 0$, $D(x) = 1$. Por ejemplo, D puede *diagonalizar* en tiempo $O(n^3)$ y definir una función que no es computable por ninguna máquina en tiempo $O(n^2)$.

5.2. La jerarquía de tiempos no-determinísticos

La técnica de diagonalización nos permite construir un objeto distinto a muchos otros. En el caso de las máquinas determinísticas, podemos 'negar' la salida para lograrlo. Para las máquinas no-determinísticas, la situación es más complicada porque no queda como se 'niega' la salida en un tiempo razonable. Lo que sí sabemos es que si $\mathcal{L} \in \mathbf{NDTime}(T(n))$, entonces $\overline{\mathcal{L}} \in \mathbf{DTime}(2^{T(n)^c})$ para alguna constante c. Usaremos este hecho más adelante.

Recordemos de la Definición 13 que un **cómputo** de una máquina no-determinística $N = (\Sigma, Q, \delta)$ a partir de $x \in \{0,1\}^*$ es una secuencia $C_0, \ldots, C_\ell$ de configuraciones tal que 1) C_0 es inicial a partir de x, 2) C_{i+1} es la evolución de C_i en un paso dado por alguna de las 2 tuplas de δ, y 3) C_ℓ está en estado $q_{\text{sí}}$ o q_{no} (ver Figura 13). En particular, si existe tal cómputo, decimos que N con entrada x **terminó**. Ya vimos que un cómputo aceptador es uno en el que C_ℓ está en estado $q_{\text{sí}}$. Llamaremos **cómputo incompleto** a una secuencia $y = C_0, \ldots, C_\ell$ como la anterior donde

valen 1) y 2) pero no vale 3). En este caso decimos que N **no terminó** (o *todavía* no terminó) siguiendo dicho cómputo.

Recordemos también que codificamos un cómputo de N con una secuencia $y \in \{0,1\}^*$ que representa las alternativas de δ. Si N es una máquina no-determinística que corre en tiempo $T(n)$, toda cadena $y_x \in \{0,1\}^{T(|x|)}$ representa un cómputo de N a partir de x. Observar que secuencias más cortas que $T(|x|)$ son posiblemente cómputos incompletos. Por simplicidad vamos a suponer que cualquier extensión de y_x codifica el mismo cómputo que y_x.

Igual que para las máquinas determinísticas, vamos a usar una nueva codificación $(N_i)_{i\in\{0,1\}^*}$ de las máquinas no-determinísticas tal que para toda máquina no-determinística N existen infinitos i tal que $N = N_i$. El Teorema 6 sigue valiendo para esta nueva codificación de máquinas. Por comodidad, a veces vamos a usar una enumeración $(N_i)_{i\in\mathbb{N}}$ de las máquinas no-determinísticas indexadas por números naturales definida de esta manera: para $i \in \mathbb{N}$, N_i representa N_z, donde z es la representación de i en binario sin el 1 inicial.

Teorema 15. *Si f, g son construibles en tiempo, no decrecientes y cumplen que $f(n+1) = o(g(n))$, entonces $\mathbf{NDTime}(f(n)) \subsetneq \mathbf{NDTime}(g(n))$.*

Demostración. Sea $(N_i)_{i\in\mathbb{N}}$ una enumeración de todas las máquinas no-determinísticas. Definimos la máquina no-determinística N con entrada x así:

si x no es de la forma 1^i0y, rechazar
si no, supongamos $x = 1^i0y$
  si $|y| < g(i)$,
    simular no-determinísticamente *$N_i(x0)$ y $N_i(x1)$ por $g(|x|)$ pasos* (*)
    si alguna no terminó: rechazar
    si las dos terminaron: aceptar sii $N_i(x0)$ acepta y $N_i(x1)$ acepta
  si $|y| = g(i)$,
    simular $N_i(1^i0)$ siguiendo el cómputo codificado por y (**)
    si no terminó: rechazar
    si terminó: aceptar sii y es no aceptador
  si $|y| > g(i)$, rechazar

La simulación (*) la realizamos gracias a la existencia de la máquina no-determinística universal NU del Teorema 6. Como N es no-determinística, podemos hacer la simulación de (*) con $NU(i, x0)$ y $NU(i, x1)$ en tiempo $O(g(|x|))$. Podemos hacer la simulación de (**) corriendo $NU(i, 1^i0)$ pero solamente considerando el cómputo codificado por y en tiempo $O(|y|)$. Es claro que N corre en tiempo $O(g(n))$, es decir que $\mathcal{L}(N) \in \mathbf{NDTime}(g(n))$.

Por el absurdo, supongamos que $\mathcal{L}(N) \in \mathbf{NDTime}(f(n))$. Hay una máquina no-determinística N' tal que $\mathcal{L}(N') = \mathcal{L}(N)$ y N' con entrada x suficientemente larga termina en a lo sumo $c \cdot f(|x|)$ pasos. Como $f(n+1) = o(g(n))$, para todo n suficientemente grande, $c \cdot f(n+1) \leq g(n)$. Tomemos i suficientemente grande tal que 1) $N' = N_i$ (esto está garantizado por la nueva codificación de máquinas no-determinísticas) y 2) para todo $x \in \{0,1\}^*$, si $|x| > i$ entonces $N_i(x0)$ y $N_i(x1)$ terminan en tiempo no-determinístico $g(|x|)$ (notar que $|x0| = |x1| = |x| + 1$ y N_i corre en tiempo $c \cdot f(n)$). Analizando la salida de N con entrada 1^i0 y usando en cada paso que $\mathcal{L}(N_i) = \mathcal{L}(N') = \mathcal{L}(N)$, tenemos el siguiente absurdo:

$$
\begin{aligned}
1^i0 \in \mathcal{L}(N) \quad &\text{sii} \quad \forall y \in \{0,1\}^1 \; 1^i0y \in \mathcal{L}(N_i) \\
&\text{sii} \quad \forall y \in \{0,1\}^2 \; 1^i0y \in \mathcal{L}(N_i) \\
&\dots \\
&\text{sii} \quad \forall y \in \{0,1\}^{g(i)} \; 1^i0y \in \mathcal{L}(N_i) \\
&\text{sii} \quad N_i(1^i0) \text{ rechaza siguiendo todos los cómputos } y \text{ tal que } |y| = g(i) \qquad (15) \\
&\text{sii} \quad 1^i0 \notin \mathcal{L}(N_i) = \mathcal{L}(N)
\end{aligned}
$$

Notemos que la equivalencia en (15) usa que $N_i(1^i0)$ termina en tiempo no-determinístico $c \cdot f(i+1) \leq g(i)$, de modo que cualquier $y \in \{0,1\}^{g(i)}$ representa un cómputo (no incompleto) de N_i con

entrada 1^i0; es decir N_i con entrada 1^i0 seguro que terminó siguiendo el cómputo $y \in \{0,1\}^{g(i)}$. La idea de esta demostración fue tomada de [2]. □

Ejemplo 13. $\mathbf{NDTime}(n^c) \subsetneq \mathbf{NDTime}(n^d)$ si $c < d$.

5.3. El Teorema de Ladner

Muchos problemas **NP** terminan siendo **P** o **NP-completos**. Surge la pregunta de si esto le pasará a todos los problemas **NP** o si habrá problemas **NP-intermedios**, que son problemas en **NP** que no son **NP-completos** y que tampoco son **P**. Desde luego, para esto hace falta la hipótesis de que $\mathbf{P} \neq \mathbf{NP}$.

Teorema 16 (Ladner)**.** *Si* $\mathbf{P} \neq \mathbf{NP}$ *entonces existe* $\mathcal{L}$ *tal que* $\mathcal{L} \in \mathbf{NP}$, $\mathcal{L} \notin \mathbf{P}$, *y* $\mathcal{L} \notin$ **NP-completo**.

Demostración. Como siempre, sea M_i la i-ésima máquina determinística. Para simplificar, escribiremos ψ en lugar de $\langle\psi\rangle$, donde ψ será una fórmula booleana (ver §4.1). Definimos el problema SAT_H relativo a la función $H : \mathbb{N} \to \mathbb{N}$ de esta manera:

$$\mathsf{SAT}_H = \{\psi 01^{n^{H(n)}} : \psi \in \mathsf{SAT}, n = |\psi|\}.$$

Notemos que cuando H crece relativamente rápido, por ejemplo cuando $H(n) = n$, entonces $\mathsf{SAT}_H \in \mathbf{P}$: si $x_\psi = \psi 01^{|\psi|^{|\psi|}}$ entonces $|x_\psi| \geq 2^{|\psi|}$ para toda fórmula booleana ψ salvo finitos casos. Dada x_ψ decidimos si ψ es satisfacible en tiempo $O(2^{|\psi|}) = O(|x_\psi|)$. Entonces, la entrada con la información de ψ es tan larga (tiene un 0 y luego suficientes 1s después de ψ) que el algoritmo de fuerza bruta para saber si ψ es satisfacible o no se vuelve lineal. En cambio, cuando H es constante, el hecho de tener algunos bits adicionales en la entrada no cambian lo que ya sabemos y $\mathsf{SAT}_H \in$ **NP-completo**.

En la demostración que sigue, vamos a elegir una H adecuada: H tiende a infinito pero crece lentamente. Eso nos va a permitir probar que SAT_H es **NP**-intermedio. Definimos H de esta manera:

$$H(n) = \begin{cases} \text{mínimo } i < \log\log n \text{ tal que para todo } x \in \{0,1\}^{\leq \log n}, & \\ M_i(x) = \chi_{\mathsf{SAT}_H}(x) \text{ en a lo sumo } i \cdot |x|^i \text{ pasos} & \text{si existe tal } i \\ \log\log n & \text{caso contrario} \end{cases}$$

Observemos que SAT_H y H están definidas por recursión mutua, pero están bien definidas: por un lado $H(n)$ usa $\mathsf{SAT}_H(x)$ para palabras x de longitud $\leq \log n$; por otro, $\mathsf{SAT}_H(x)$ usa $H(n)$ para $n < |x|$, donde $x = \psi 01^{n^{H(n)}}$ y $n = |\psi|$.

Ejercicio 7. Probar que H es computable en tiempo $O(n^3)$.

Sea $f(x)$ la función que devuelve ψ si x es de la forma $\psi 01^{n^{H(n)}}$ con $n = |\psi|$ o $p \wedge \neg p$ en caso contrario. Es claro $\mathsf{SAT}_H \leq_\mathrm{p} \mathsf{SAT}$ vía f. Como $\mathsf{SAT} \in \mathbf{NP}$ y **NP** está cerrado por $\leq_\mathrm{p}$, concluimos que $\mathsf{SAT}_H \in \mathbf{NP}$.

Proposición 12. $\mathsf{SAT}_H \in \mathbf{P} \implies \exists c\ \forall n\ H(n) \leq c$.

Demostración. Supongamos que la máquina determinística M decide SAT_H en tiempo $c \cdot n^c$. Existe $i > c$ tal que $\mathcal{L}(M) = \mathcal{L}(M_i)$. Entonces para todo $n \geq 2^{2^i}$, tenemos que $H(n) \leq i$. Luego

$$H(n) \leq \text{máx}\left(\{H(m) : m < 2^{2^i}\} \cup \{i\}\right),$$

y esto termina la demostración de la Proposición. □

Proposición 13. $\exists c\ \exists^\infty n\ H(n) \leq c \implies \mathsf{SAT}_H \in \mathbf{P}$.

Demostración. Supongamos $\exists c\, \exists^\infty n\; H(n) \leq c$. Entonces $\exists i\, \exists^\infty n\; H(n) = i$. Veamos que M_i decide SAT_H en tiempo $i \cdot n^i$. Supongamos que M_i no decide SAT_H. Existe x tal que $M_i(x) \neq \chi_{\mathsf{SAT}_H}(x)$. Entonces, por definición de H, tenemos que $\forall n > 2^{|x|}\; H(n) \neq i$ y esto es un absurdo. Si suponemos que M_i no corre en tiempo $i \cdot n^i$, razonamos análogamente y llegamos también a un absurdo. Esto termina la demostración de la Proposición. □

Notemos que el contra-recíproco de la Proposición 13 es $\mathsf{SAT}_H \notin \mathbf{P} \implies \lim_{n\to\infty} H(n) = \infty$.

Veamos que $\mathsf{SAT}_H \notin \mathbf{P}$. Supongamos $\mathsf{SAT}_H \in \mathbf{P}$. Por la Proposición 12, $\exists c\; \forall n\; H(n) \leq c$. Tenemos que

$$\begin{aligned}\mathsf{SAT}_H &= \{\psi 01^{n^{H(n)}} : \psi \in \mathsf{SAT}, n = |\psi|\} \\ &= \{\psi 0 \underbrace{1 \ldots\ldots 1}_{|\psi|^{H(|\psi|)} \leq |\psi|^c} : \psi \in \mathsf{SAT}\}.\end{aligned}$$

Pero entonces un algoritmo de tiempo polinomial para SAT_H se puede usar para SAT, por lo que $\mathsf{SAT} \in \mathbf{P}$, es decir $\mathsf{SAT}_H \leq_\mathrm{p} \mathsf{SAT}$. Como SAT es **NP-completo** concluimos que $\mathbf{P} = \mathbf{NP}$ y esto es un absurdo porque contradice la hipótesis del Teorema que queremos probar.

Veamos que $\mathsf{SAT}_H \notin$ **NP-completo**. Por el absurdo, supongamos $\mathsf{SAT}_H \in$ **NP-completo**. Entonces hay una reducción polinomial f de SAT a SAT_H tal que

$$\begin{aligned}\psi \in \mathsf{SAT} \quad &\text{sii} \quad f(\psi) \in \mathsf{SAT}_H \\ &\text{sii} \quad f(\psi) \text{ es de la forma } F_\psi 01^{|F_\psi|^{H(|F_\psi|)}} \text{ con } F_\psi \in \mathsf{SAT}, \end{aligned} \tag{16}$$

para toda fórmula booleana ψ. Aquí F_ψ es (la codificación de) alguna fórmula booleana. Supongamos que f es computable en tiempo $c \cdot n^c$ por una cierta máquina determinística. Como esa máquina no puede escribir más que $c \cdot n^c$ símbolos en la salida, $|f(x)| \leq c \cdot |x|^c$. Luego para toda fórmula booleana ψ tenemos que

$$\begin{aligned}|f(\psi)| &= |F_\psi 01^{|F_\psi|^{H(|F_\psi|)}}| \\ &= |F_\psi| + 1 + |F_\psi|^{H(|F_\psi|)} \leq c \cdot |\psi|^c. \end{aligned} \tag{17}$$

Proposición 14. *Existe k tal que para toda fórmula booleana ψ, $|\psi| > k \implies |F_\psi| < |\psi|$.*

Demostración. Supongamos que vale lo contrario. Para todo k, existe una fórmula booleana ψ tal que $|\psi| > k$ y $|F_\psi| \geq |\psi| > k$. Como $\mathsf{SAT}_H \notin \mathbf{P}$, por el contrarecíproco de la Proposición 13, tenemos que $\lim_{n\to\infty} H(n) = \infty$. Entonces

$$\exists k'\; \forall n > k'\; n^{H(n)} > c \cdot n^c. \tag{18}$$

Sea ψ tal que $|F_\psi| \geq |\psi| > k'$. Luego

$$\begin{aligned} c \cdot |\psi|^c &\leq c \cdot |F_\psi|^c && \text{(pues } |F_\psi| \geq |\psi|) \\ &< |F_\psi|^{H(|F_\psi|)} && \text{(por (18) y el hecho de que } |F_\psi| > k') \\ &\leq c \cdot |\psi|^c, && \text{(por (17))} \end{aligned}$$

y esto es un absurdo, que termina la demostración de la Proposición. □

Tomemos el k de la Proposición 14 y analicemos el siguiente algoritmo recursivo G:

entrada: ψ
si $|\psi| \leq k$,
  devolver 'sí' si $\psi \in \mathsf{SAT}$ y 'no' en caso contrario (hay finitos casos porque k es constante)
si no,
  computar $f(\psi)$
  si $f(\psi)$ no es de la forma $F_\psi 01^{|F_\psi|^{H(|F_\psi|)}}$, devolver 'no'

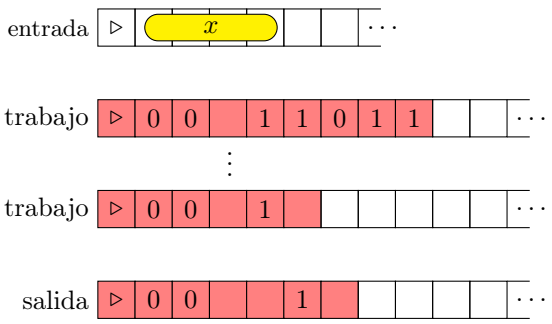


Figura 18: El espacio usado por un cómputo es la máxima cantidad de celdas visitadas (celdas en rojo). Se cuentan todas las celdas salvo las de la cinta de entrada.

si no, devolver $G(F_\psi)$ (notar que por la Proposición 14, $|F_\psi| < |\psi|$)

Es claro que G con entrada de tamaño n realiza a lo sumo n llamados recursivos y, como f es computable en tiempo polinomial, G corre en tiempo polinomial. Además, por (16) tenemos que $\psi \in \mathsf{SAT}$ sii $G(\psi) \in \mathsf{SAT}$. Entonces $\mathbf{P} = \mathbf{NP}$ y esto contradice la hipótesis. □

6. Espacio usado por un cómputo

Hasta ahora nos centramos en el recurso del tiempo que tomaba un cómputo. En esa sección nos vamos a centrar en otro recurso importante: la memoria. En el formalismo de cómputo que estudiamos, las máquinas, la memoria se formaliza con la cantidad de celdas que visita la máquina a lo largo del cómputo en las cintas de trabajo y salida. En la Figura 18, las cabezas solo pasan por las celdas marcadas en rojo; el espacio usado por ese cómputo será, entonces, la cantidad de celdas en rojo. Notemos que no se cuentan las celdas de la cinta de entrada, que seguramente son visitadas muchas veces a lo largo del cómputo, con la cabeza de la cinta de entrada yendo y viniendo varias veces, buscando partes de la entrada.

Definición 18. Dada una máquina determinística M y $x \in \{0,1\}^*$ el **espacio** que usa M con entrada $x \in \{0,1\}^*$ es la cantidad de celdas por las que alguna vez pasó la cabeza en las cintas de trabajo y de salida a lo largo del cómputo de M a partir de x.

Dada una máquina no-determinística N y $x \in \{0,1\}^*$ el **espacio** que usa N con entrada x es la cantidad de celdas por las que alguna vez pasó la cabeza en las cintas de trabajo y de salida a lo largo de *todos* los cómputos de N a partir de x.

Remarquemos de nuevo que no contamos en el espacio las celdas de entrada (que en general van a tener que ser leídas todas) porque en algunos casos estudiaremos algunos problemas simples que pueden resolverse en espacio sublineal.

Definición 19. La máquina (determinística o no-determinística) **usa espacio** $S(n)$ si para toda entrada $x \in \{0,1\}^*$, el espacio que usa M con entrada x es a lo sumo $S(|x|)$. M **usa espacio** $O(S(n))$ si existe una constante c tal que para todo x, salvo finitos, M con entrada x usa espacio a lo sumo $c \cdot S(|x|)$. Una función f es computable **en espacio** $S(n)$, [**en espacio** $O(S(n))$] si existe una máquina determinística que computa f y usa espacio $S(n)$ [usa espacio $O(S(n))$].

Clase de complejidad: $\textsc{Space}(S(n)), \textsc{NSpace}(S(n))$

$\textsc{Space}(S(n))$ es la clase de lenguajes $\mathcal{L}$ tal que existe una máquina determinística M tal que M decide $\mathcal{L}$ y M usa espacio $O(S(n))$.

$\textsc{NSpace}(S(n))$ es la clase de lenguajes $\mathcal{L}$ tal que existe una máquina no-determinística N

tal que N decide $\mathcal{L}$ y N usa espacio $O(S(n))$

Definición 20. Una función $S : \mathbb{N} \to \mathbb{N}$ es **construible en espacio** si la función $1^n \mapsto [S(n)]$ es computable en espacio $O(S(n))$

Ejemplo 14. $\log n$, n, n^2, 2^n son funciones construibles en espacio.

Cuando estudiamos el recurso del tiempo, no eran interesantes las funciones $T(n) < n$ cuando medíamos el tiempo, porque se espera que las máquinas al menos tengan que leer la cinta de entrada, y eso lleva tiempo n. Cuando medimos espacio, la situación es distinta. Como no medimos el espacio de la entrada, tiene sentido estudiar funciones de espacio $S(n) < n$. Sin embargo, a veces vamos a requerir que $S(n) \geq \log n$, dado que queremos poder 'recordar' cualquier *índice* de la cinta de entrada (la entrada tiene largo n, entonces representamos cualquier $i \leq n$ con una palabra de $\log n$ bits).

Proposición 15. $\textbf{DTime}(S(n)) \subseteq \textbf{Space}(S(n))$.

Demostración. En cada paso, una máquina determinística solo puede usar una celda más de cada cinta. Si M corre en tiempo $O(S(n))$, entonces M usa espacio $O(S(n))$. □

El siguiente resultado es trivial de las definiciones.

Proposición 16. $\textbf{Space}(S(n)) \subseteq \textbf{NSpace}(S(n))$.

Definimos un grafo que va a ser una herramienta importante para los resultados que siguen.

Definición 21. Sea $S : \mathbb{N} \to \mathbb{N}$ y sea M una máquina (determinística o no-determinística) que usa espacio $S(n)$. El **grafo de configuraciones** de M con entrada x, notado $G_{M,x}$ es un grafo dirigido tal que el conjunto de vértices son las configuraciones de M con la información de las cintas de trabajo de hasta $S(|x|)$ celdas (y no más allá de la posición $S(|x|)$) en cada cinta de trabajo y en la de salida, y hay una arista de C a C' si C' es una evolución en un paso de C dado por M.

Si M es una máquina determinística o no-determinística, M acepta $x \in \{0,1\}^*$ si y solo si existe un camino en $G_{M,x}$ desde la configuración inicial de M para x hasta alguna configuración final. Ver Figura 19.

Como siempre, estamos interesados en máquinas que deciden lenguajes. Si M es determinística, $G_{M,x}$ tiene *out-degree* 1; podemos suponer que M borra todas las cintas de trabajo y deja todas las cabezas en la primera celda antes de entrar a q_f. Así hay un *único* estado final $C_{\text{sí}}$ aceptador: es el que tiene el estado en q_f y en la cinta de salida hay un 1 seguido de blancos. En cambio, si N es no-determinística, $G_{N,x}$ tiene *out-degree* ≤ 2; podemos suponer que N borra todas las cintas de trabajo y salida y deja todas las cabezas en la primera celda antes de entrar a $q_{\text{sí}}$. Así hay un único estado final $C_{\text{sí}}$ aceptador.

Proposición 17. *Sea M una máquina (determinística o no-determinística) que usa espacio $S(n)$ tal que $S(n) \geq \log n$. Existe una constante c tal que para todo $x \in \{0,1\}^*$, cada vértice de $G_{M,x}$ se puede describir usando $c \cdot S(|x|)$ bits (c depende de M). Luego, $G_{M,x}$ tiene $2^{c \cdot S(|x|)}$ nodos.*

Demostración. Sea $M = (\Sigma, Q, \delta)$ con k cintas de trabajo. Codificamos cada configuración de un cómputo de M con entrada $x \in \{0,1\}^*$ de la siguiente manera. Numeramos los estados de Q y usamos la codificación usual (codificamos q con $\langle q \rangle$). Supongamos que T_i, la i-ésima cinta de trabajo, tiene el contenido $\triangleright\, b_1^i\, b_2^i\, \ldots b_{S(|x|)}^i$. La posición de la cabeza es un número $0 \leq j_i \leq S(|x|)$. Podemos codificar el contenido de T_i y la posición de la cabeza con

$$\langle T_i \rangle = 0b_1^i \ldots 0b_{j_i-1}^i\ 10\ 0b_{j_i}^i \ldots 0b_{S(|x|)}^i.$$

Hacemos lo mismo con la cinta de salida Z. Para la cinta de entrada E solo codificamos la *posición* p de la cabeza y la codificamos con $\langle E \rangle = \langle p \rangle$ y por lo tanto $|E| = O(\log |x|)$. Notemos que no usamos la misma configuración que para las cintas de trabajo o salida porque podría ser $n > S(n)$.

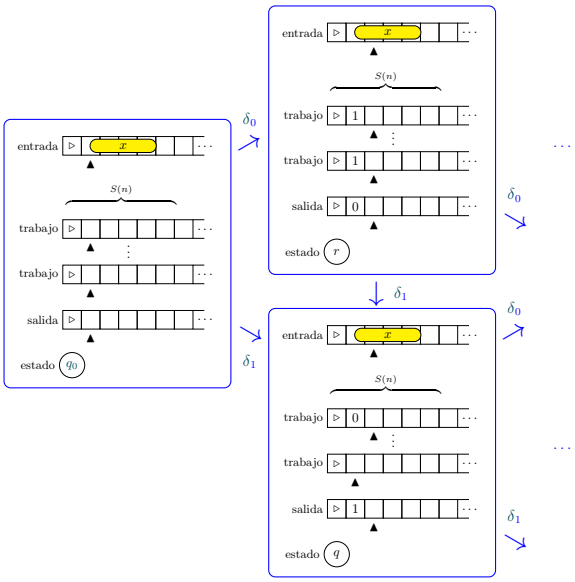


Figura 19: Un grafo de configuraciones $G_{N,x}$. Cada nodo representa un estado del cómputo de la máquina no-determinística N con entrada x, donde N usa espacio $S(n)$. En cada nodo alcanza con representar el contenido de las cintas de trabajo y salida hasta la posición $S(|x|)$, porque nunca se van a usar más allá de esa posición. Las aristas representan las posibles evoluciones en un paso (a lo sumo dos si se trata de una máquina no-determinística).

Si $n = |x|$, entonces cada configuración C se codifica con $\langle C \rangle = \langle \langle q \rangle, \langle E \rangle, \langle T_1 \rangle, \ldots, \langle T_k \rangle, \langle Z \rangle \rangle$ y recordando que $S(n) \geq \log n$ tenemos que

$$|C| = d \cdot ((k+1) \cdot S(n) + \log n) = c \cdot S(n).$$

Aquí c, d dependen de M pero son independientes de n. Así, codificamos cada configuración C con una cadena sobre el alfabeto binario, por lo que hay $2^{c \cdot S(n)}$ posibles configuraciones (para c dependiente de M pero independiente de x). □

Proposición 18. *Si S es construible en espacio,* $\mathbf{NSpace}(S(n)) \subseteq \mathbf{DTime}(2^{O(S(n))})$.

Demostración. Sea N una máquina no-determinística que decide $\mathcal{L}$ en espacio $O(S(n))$. Definimos la máquina determinística M con entrada x:

> *Construir $G_{N,x}$ (toma tiempo $2^{O(S(n))}$). Usar BFS para decidir si existe un camino entre la configuración inicial de N a partir de x y la (única) configuración final. Si existe tal camino, escribir 1 en la salida; si no escribir 0. Luego pasar a q_f.*

Para cualquier grafo $G = (V, E)$, BFS corre en tiempo $O(|V| + |E|)$. Como el *out-degree* de cada vértice de $G_{N,x}$ es a lo sumo 2, la cantidad de aristas de $G_{N,x}$ es $O(2^{O(S(|x|))})$. Luego BFS sobre $G_{N,x}$ toma tiempo $O(2^{O(S(|x|))})$. Entonces $\mathcal{L}(M) = \mathcal{L}$ y M corre en tiempo $O(2^{O(S(n))})$. □

6.1. Espacio polinomial: PSpace y NPSpace

Clase de complejidad: PSpace, NPSpace

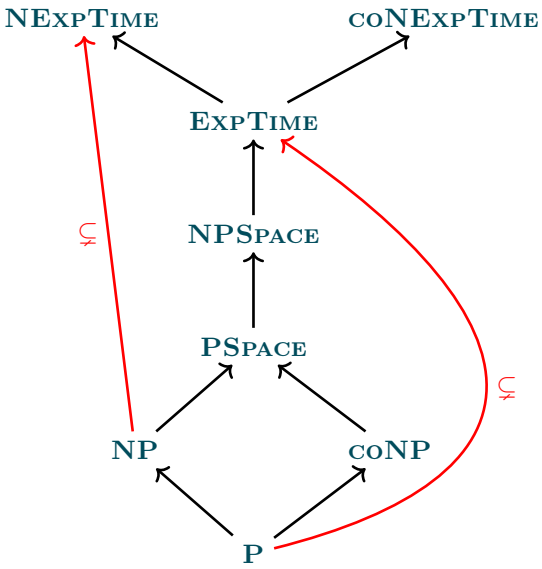


Figura 20: Las clases de complejidad vistas hasta ahora. Por el Teorema 14, $\mathbf{P} \subsetneq \mathbf{ExpTime}$ y por el Teorema 15, $\mathbf{NP} \subsetneq \mathbf{NExpTime}$.

$$\mathbf{PSpace} = \bigcup_{c>0} \mathbf{Space}(n^c)$$
$$\mathbf{NPSpace} = \bigcup_{c>0} \mathbf{NSpace}(n^c)$$

Observar que $\mathbf{PSpace} \subseteq \mathbf{NPSpace} \subseteq \mathbf{ExpTime}$.

Proposición 19. $\mathbf{NP} \subseteq \mathbf{PSpace}$.

Demostración. Supongamos $\mathcal{L} \in \mathbf{NP}$. Existe una máquina determinística M y un polinomio $p : \mathbb{N} \to \mathbb{N}$ tal que M corre en tiempo polinomial y para todo $x \in \{0,1\}^*$ tenemos que $x \in \mathcal{L}$ sii existe $u \in \{0,1\}^{p(|x|)}$ tal que $M(\langle x, u\rangle) = 1$ Definimos una máquina determinística M' que con entrada x hace esto:

> *Simula $M(\langle x, u\rangle)$ para cada $u \in \{0,1\}^{p(|x|)}$. Cada u lo escribe en las* mismas *$p(|x|)$ celdas (reutiliza el espacio). Si encuentra u tal que $M(\langle x, u\rangle) = 1$, escribe 1 en la salida. Si no, escribe 0. Luego pasa a q_f.*

M' decide $\mathcal{L}$; corre en tiempo exponencial pero usa espacio $p(|x|)$ y el que usa M con entrada $\langle x, u\rangle$, que es polinomial. Luego $\mathcal{L} \in \mathbf{PSpace}$. □

La Figura 20 muestra la relación entre las clases de complejidad que vimos hasta ahora.

Al igual que para los tiempos, existe una jerarquía de espacio. El siguiente resultado es análogo al Teorema 14.

Teorema 17. *Si f, g son construibles en espacio y cumplen que $f(n) = o(g(n))$ entonces*

$$\mathbf{Space}(f(n)) \subsetneq \mathbf{Space}(g(n)).$$

Ejercicio 8. Demostrar el Teorema 17. Es la misma idea que para la jerarquía de tiempos determinísticos (Teorema 14), solo que podemos simular cualquier máquina con un un factor constante de costo extra en lugar de logarítmico.

6.2. Fórmulas booleanas cuantificadas y el Teorema de Savitch

Definición 22. Una **fórmula booleana cuantificada** (o QBF) es una expresión de la forma

$$Q_1 x_1 \; Q_2 x_2 \ldots Q_n x_n \; \varphi(x_1, x_2, \ldots, x_n) \tag{19}$$

donde cada Q_i es un cuantificador $\forall$ o un cuantificador $\exists$, y $\varphi(x_1, x_2, \ldots, x_n)$ es una fórmula booleana con variables entre $x_1, \ldots, x_n \in \mathsf{PROP}$ y posiblemente con constantes $0, 1$ de la gramática dada en (6).

Podemos extender las QBFs con la incorporación de variables libres:

$$\psi = \psi(y_1, \ldots, y_m) = Q_1 x_1 \; Q_2 x_2 \ldots Q_n x_n \; \varphi(x_1, x_2, \ldots, x_n, y_1, \ldots, y_m) \tag{20}$$

donde cada Q_i es un cuantificador $\forall$ o un cuantificador $\exists$, donde $x_1, \ldots, x_n$ son **variables cuantificadas**, donde $y_1, \ldots, y_m$ $(m \geq 0)$ son **variables libres** (es decir, no cuantificadas) de ψ y donde $\psi(x_1, x_2, \ldots, x_n, y_1, \ldots, y_m)$ es una fórmula booleana con variables entre $x_1, \ldots, x_n, y_1, \ldots, y_m$. Usar ψ o $\psi(y_1, \ldots, y_m)$ es indistinto; la segunda notación solo enfatiza que las variables libres de ψ están entre $y_1, \ldots, y_m$. Es solo una cuestión de notación pero representan la misma QBF.

Cuando hablemos de QBF a secas estaremos pensando que 1) siempre están en forma *prenexa*, que quiere decir que todos los cuantificadores están adelante, y 2) no contienen variables libres. Cuando queramos referirnos a una QBF como la de (20), hablaremos de QBF **posiblemente con variables libres**.

Igual que para las fórmulas booleanas de la §4.1, las QBF posiblemente con variables libres toman un valor verdadero o falso dada una valuación $v : \mathsf{PROP} \to \{0, 1\}$. Los cuantificadores $\forall$ o $\exists$ tienen la semántica usual de 'para todo' y 'existe', pero los valores que toman las variables x_i son *booleanos*, es decir pueden tomar solamente valor 0 (falso) o 1 (verdadero). Más formalmente, extendemos la noción de verdad de una fórmula booleana con constantes de §4.1 con:

- $v \models \forall x \; \varphi$ si $v(x \mapsto 0) \models \varphi$ y $v(x \mapsto 1) \models \varphi$
- $v \models \exists x \; \varphi$ si $v(x \mapsto 0) \models \varphi$ o $v(x \mapsto 1) \models \varphi$

donde para $b \in \{0, 1\}$, $v(x \mapsto b)$ representa una valuación v' tal que $v'(y) = v(y)$ si $y \neq x$ y $v'(x) = b$. Como antes usamos la notación de $\models \varphi$ para referirnos a que $v \models \varphi$ para toda valuación v. Como las QBF no tienen variables libres (en lógica se llaman **sentencias**), su verdad o falsedad es independiente de cualquier valuación.

Ejemplo 15. La QBF $\psi = \forall x_1 \exists x_2 \forall x_3 \; (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$ es falsa: si x_1 es verdadera, entonces x_3 tiene que ser verdadera, pero x_3 está cuantificada universalmente.

Como con las fórmulas booleanas, representamos valuaciones (parciales) $\{y_1, \ldots, y_m\} \to \{0, 1\}$ con palabras v en $\{0, 1\}^m$, notamos $v \models \psi(y_1, \ldots, y_m)$ cuando ψ es verdadera para v, y notamos $v \not\models \psi(y_1, \ldots, y_m)$ cuando ψ es falsa para v.

Ejemplo 16. Para $\psi(y_1, y_2) = \exists x \; (x \leftrightarrow y_1) \wedge (x \leftrightarrow y_2)$ (acá $u \leftrightarrow v$ es una abreviatura para $(u \wedge v) \vee (\neg u \wedge \neg v)$), tenemos que $00 \models \psi(y_1, y_2)$, $11 \models \psi(y_1, y_2)$, $01 \not\models \psi(y_1, y_2)$.

Al igual que con las fórmulas booleanas de §4.1, las QBF posiblemente con variables libres se pueden codificar con palabras de $\{0, 1\}^*$ extendiendo la definición de (7):

$$\langle \exists x_i \; \varphi \rangle = 0101 \langle x_i \rangle \langle \varphi \rangle \qquad \langle \forall x_i \; \varphi \rangle = 0110 \langle x_i \rangle \langle \varphi \rangle \tag{21}$$

(El hecho de que usar x_i en vez de p_i es indistinto; las variables son siempre elementos de PROP y ya sabemos de (7) cómo codificarlas.)

Problema: ***True Quantified Boolean Formula***

$$\mathsf{TQBF} = \{\langle\psi\rangle : \psi \text{ es una QBF tal que } \models \psi\}$$

Supongamos la QBF

$$\psi = Q_1 x_1 \overbrace{Q_2 x_2 \dots Q_n x_n \ \varphi(x_1, x_2, \dots, x_n)}^{\rho}.$$

Para $b \in \{0,1\}$, definimos $\psi \restriction_{x_1=b}$ como el resultado de reemplazar en ρ todas las ocurrencias de x_1 por la constante b. Observar que $\psi \restriction_{x_1=b}$ también es una QBF y que

- si $Q_1 = \forall$ entonces $\models \psi$ sii $\models \psi \restriction_{x_1=0}$ y $\models \psi \restriction_{x_1=1}$
- si $Q_1 = \exists$ entonces $\models \psi$ sii $\models \psi \restriction_{x_1=0}$ o $\models \psi \restriction_{x_1=1}$

Estas reglas nos permiten eliminar cuantificadores.

Teorema 18. $\mathsf{TQBF} \in$ **PSpace**.

Demostración. Definimos una función recursiva F que toma como entrada una (codificación de una) QBF como la de (19) y devuelve como salida 1 si $\models \psi$ o 0 en caso contrario.

si $n = 0$, ψ no tiene variables (solo tiene constantes $0, 1$)
  decidir si es verdadera o falsa *(tiempo $O(|\psi|)$)*
si no, ψ es de la forma $\psi = Qx\ \rho$
  computar $i = F(\psi \restriction_{x=0})$ y $j = F(\psi \restriction_{x=1})$ *(se reutiliza el espacio en los dos llamados)*
  solo nos quedamos con los bits i, j
  si $Q = \forall$, devolver 1 si $i = j = 1$; si no, devolver 0
  si $Q = \exists$, devolver 1 si $i = 1$ o $j = 1$; si no, devolver 0

El algoritmo necesita espacio $O(|\psi|)$ para escribir $\psi \restriction_{x=b}$ y en cada llamado decrementamos el tamaño de la entrada. En la i-ésima llamada recursiva, $(i \leq |\psi|)$ la entrada tiene tamaño $O(i)$ y necesita $O(i)$ bits para representar $\psi \restriction_{x=0}$ y $\psi \restriction_{x=1}$. En total usa espacio $O(|\psi|^2)$. □

Podemos considerar también una gramática todavía más general:

$$\varphi ::= x \mid 0 \mid 1 \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x\ \varphi \mid \exists x\ \varphi$$

con la semántica que ya dimos. Llamémoslas **QBF generalizadas**. Observemos que no son QBFs porque no están en forma prenexa y además porque pueden tener variables libres. Se codifican con palabras de $\{0,1\}^*$ como vimos en (21).

Ejemplo 17. $\psi(y) = \exists x_1\ (\forall x_2\ (y \wedge \neg x_1 \wedge x_2) \vee \forall x_3\ (\neg y \vee x_1 \vee x_3))$ no es una QBF pero es una QBF generalizada.

Proposición 20. *En tiempo polinomial se puede transformar cualquier QBF generalizada $\psi(y_1, \dots, y_m)$ en una QBF $\psi'(y_1, \dots, y_m)$ posiblemente con variables libres que es equivalente, es decir, tal que para todo $v \in \{0,1\}^m$*

$$v \models \psi(y_1, \dots, y_m) \qquad \textit{sii} \qquad v \models \psi'(y_1, \dots, y_m).$$

Ejercicio 9. Demostrar la Proposición 20. Idea: renombrar variables cuantificadas si hace falta y aplicar reglas:

$$\neg\forall x\ \psi = \exists x\ \neg\psi$$
$$\neg\exists x\ \psi = \forall x\ \neg\psi$$
$$\psi * (Qx\ \varphi(x)) = Qx\ (\varphi * \psi) \qquad (Q \in \{\forall, \exists\}, * \in \{\wedge, \vee\},\ \psi \text{ no contiene a } x \text{ libre})$$

Problema: Chequeo de QBF generalizada

$$\mathsf{CHECKQBF} = \{\langle \psi, v\rangle : \begin{array}{l}\psi \text{ es una QBF generalizada con } m \text{ variables libres,} \\ v \in \{0,1\}^m \text{ y } v \models \psi\end{array}\}$$

Proposición 21. $\mathsf{CHECKQBF} \leq_{\mathrm{p}} \mathsf{TQBF}$.

Demostración. Sea ψ una QBF generalizada con variables libres $y_1, \ldots, y_m$ y sea $v \in \{0,1\}^m$. Primero llevamos ψ a una ψ' equivalente pero en forma prenexa (ψ' también tiene variables libres $y_0, \ldots, y_{m-1}$). Por la Proposición 20 esto toma tiempo polinomial. Luego definimos ψ'' como el resultado de reemplazar cada variable y_j en ψ' por la constante $v(j)$; nos queda una QBF tal que[13]

$$\begin{array}{lll}\langle \psi, v\rangle \in \mathsf{CHECKQBF} & \text{sii} & v \models \psi(y_1, \ldots, y_m) \\ & \text{sii} & \models \psi'' \\ & \text{sii} & \psi'' \in \mathsf{TQBF}\end{array}$$

La transformación $\langle \psi, v\rangle \mapsto \psi''$ es computable en tiempo polinomial y esto prueba que $\mathsf{CHECKQBF} \leq_{\mathrm{p}} \mathsf{TQBF}$. □

Proposición 22. *Sea M una máquina (determinística o no-determinística) que usa espacio $S(n) \geq \log n$ y sea c una constante tal que para todo $x \in \{0,1\}^*$ cualquier configuración C de un cómputo de M con entrada x se representa con $|\langle C\rangle| = c \cdot S(|x|)$ bits. Dado $x \in \{0,1\}^*$, podemos computar en tiempo polinomial en $S(n)$ una fórmula $\varphi_{M,x}(\bar{s}, \bar{t})$ en CNF con variables libres $\bar{s} = s_1, \ldots, s_{c\cdot S(|x|)}$, $\bar{t} = t_1, \ldots, t_{c\cdot S(|x|)}$ tal que $\langle C\rangle\langle C'\rangle \models \varphi_{M,x}(\bar{s}, \bar{t})$ sii hay una flecha de C a C' en $G_{M,x}$.*

Idea de la demostración. Es parecido a lo que hicimos con la demostración del Teorema 11. En (14), construíamos la fórmula booleana ψ_3^i que expresaba que la mini-configuración z_i evolucionaba en un paso en z_{i+1}, donde $z_0 \ldots z_m$ representaba un cómputo. Esa fórmula booleana predicaba sobre mini-configuraciones. Ahora tenemos configuraciones completas, es decir, con todo el contenido de todas las cintas, de tamaño $|\langle C\rangle| = c \cdot S(|x|)$ y tenemos que expresar cuándo una configuración C, codificada con las variables $\bar{t}$, es la evolución en un paso de otra configuración C', codificada con las variables $\bar{s}$.

Recordemos la codificación de las configuraciones de la demostración de la Proposición 17. Cada cinta de trabajo codifica simultáneamente su contenido y la posición de su cabeza. Supongamos que M tiene una sola cinta de trabajo. Alcanza con mirar qué pasa con la codificación de 3 bloques de 2 bits en la codificación de la cinta. Por ejemplo, supongamos que la cabeza de la cinta de entrada está leyendo b_e y que M está en el estado q. Si en la parte que codifica la cinta de trabajo de $\langle C\rangle$ (variables en $\bar{s}$) vemos una ventana $0b_1\ 10\ 0b_2$ (recodar que cada bit se codifica con una variable booleana en $\varphi_{M,x}$), con $b_i \in \{0,1\}$, sabemos que está leyendo el bit b_2 y entonces podemos determinar el contendido de la misma ventana pero en la codificación de $\langle C'\rangle$ (variables en $\bar{t}$) porque contamos con la información de δ y del símbolo leído en la entrada. Más en detalle, si $\delta(q, b_e, b_2) = (*, b_3, R, *, *)$ sabemos que la cabeza de la cinta en $\langle C'\rangle$ escribe b_3 y se mueve a la derecha; por lo tanto la ventana correspondiente en $\langle C\rangle'$ es $0b_1\ 0b_3\ 10$. Lo importante es que el comportamiento solo depende de la ventana, es decir, de variables proposicionales que participan en la descripción de esa ventana. De la misma forma, se puede expresar cómo cambia el estado, la posición de cabeza de la cinta de entrada (recordemos que no codificamos el contenido de la cinta de entrada) y el contenido y posición de la cabeza en la cinta de salida. La formula final tiene que 'deslizar' esas ventanas por todas las posiciones de las cintas en $\langle C\rangle$ y $\langle C'\rangle$. Así, la fórmula que buscamos se computa en tiempo polinomial en $S(n)$, y por lo tanto tiene tamaño también polinomial en $S(n)$. □

Ejercicio 10. Completar los detalles en la demostración de la Proposición 22.

[13] Recordemos de (20) que el uso de $(y_1, \ldots, y_m)$ en una fórmula $\rho(y_1, \ldots, y_m)$ es solo para avisar que las variables libres de ρ están entre $y_1, \ldots, y_m$; $\rho = \rho(y_1, \ldots, y_m)$.

Teorema 19. TQBF $\in$ **PSPACE-hard**.

Demostración. Sea $\mathcal{L} \in$ **PSPACE** y sea M una máquina determinística que decide $\mathcal{L}$ en espacio $S(n)$, con S un polinomio. Consideramos el grafo de configuraciones $G_{M,x}$ de la Definición 21. Cada vértice $\langle C \rangle$ de $G_{M,x}$ es la codificación con $c \cdot S(n)$ bits de una configuración, donde $n = |x|$. Sea C_0 la configuración inicial y C_f la configuración final del cómputo de M a partir de x.

Definimos fórmulas $\psi_i(\bar{s}, \bar{t})$ con variables libres $\bar{s}$, $\bar{t}$, donde $\bar{s}$ y $\bar{t}$ son tuplas de dimensión $c \cdot S(n)$ con la siguiente propiedad: $\langle C \rangle \langle C' \rangle \models \psi_i(\bar{s}, \bar{t})$ sii existe un camino dirigido de longitud a lo sumo 2^i en $G_{M,x}$ entre C y C'. Si hay un camino de C a C' en $G_{M,x}$, entonces hay uno de longitud a lo sumo $2^{c \cdot S(n)}$, porque $G_{M,x}$ tiene a lo sumo $2^{c \cdot S(n)}$ vértices.

Luego, tenemos que

$$\begin{aligned} x \in \mathcal{L} \quad &\text{sii} \quad \langle C_0 \rangle \langle C_f \rangle \models \psi_{c \cdot S(n)}(\bar{s}, \bar{t}) \\ &\text{sii} \quad \langle \psi_{c \cdot S(n)}(\bar{s}, \bar{t}), \langle C_0 \rangle \langle C_f \rangle \rangle \in \mathsf{CHECKQBF}. \end{aligned} \tag{22}$$

Entonces $\mathcal{L} \leq_{\mathrm{p}} \mathsf{CHECKQBF} \leq_{\mathrm{p}} \mathsf{TQBF}$, siempre y cuando podamos definir $\psi_{c \cdot S(n)}$ en tiempo polinomial en n.

Definimos $\psi_i(\bar{s}, \bar{t})$ por recursión en i. Para $i = 0$, definimos $\psi_0(\bar{s}, \bar{t})$ como la fórmula $\bar{s} = \bar{t} \vee \varphi_{M,x}(\bar{s}, \bar{t})$, donde $\varphi_{M,x}(\bar{s}, \bar{t})$ es la fórmula de la Proposición 22 y $\bar{s} = \bar{t}$ abrevia $\bigwedge_{i=1}^{2^{c \cdot S(n)}} (s_i \wedge t_i) \vee (\neg s_i \wedge \neg t_i)$ si tomamos que $\bar{s} = s_1, \ldots, s_{2^{c \cdot S(n)}}$ y $\bar{t} = t_1, \ldots, t_{2^{c \cdot S(n)}}$. Entonces, computamos ψ_0 en tiempo polinomial en n. Para $i > 0$ podríamos definir

$$\psi_i(\bar{s}, \bar{t}) = \exists \bar{r} \; \psi_{i-1}(\bar{s}, \bar{r}) \wedge \psi_{i-1}(\bar{r}, \bar{t})$$

pero el tamaño de ψ_i sería exponencial en i (porque duplica su longitud en cada llamado recursivo) y por lo tanto no podríamos computarla en tiempo polinomial. Esta idea no nos sirve para probar la reducción polinomial (22) que buscamos. El truco es observar que existe un camino de longitud a lo sumo 2^j entre C y C' en el grafo de configuraciones $G_{M,x}$ sii existe una configuración intermedia C'' tal que hay un camino de longitud a lo sumo 2^{i-1} entre C y C'' y uno de longitud a lo sumo 2^{j-1}entre C'' y C'. Así, definimos la fórmula

$$\psi_i(\bar{s}, \bar{t}) = \exists \bar{r} \forall \bar{u}, \bar{v} \; ((\bar{u} = \bar{s} \wedge \bar{v} = \bar{r}) \vee (\bar{u} = \bar{r} \wedge \bar{v} = \bar{t})) \to \psi_{i-1}(\bar{u}, \bar{v}) \tag{23}$$

En esta fórmula estamos abreviando $\phi \to \rho$ por $\neg\phi \vee \rho$ y $\bar{a} = \bar{b}$ por $\bigwedge_{i \leq k}(a_i \wedge b_i) \vee (\neg a_i \wedge \neg b_i)$, para $\bar{a} = a_1, \ldots, a_k$ y $\bar{b} = b_1, \ldots, b_k$. La QBF generalizada (23) expresa que hay una configuración intermedia $\bar{r}$ entre $\bar{s}$ y $\bar{t}$ tal que vale ψ_{i-1} tanto si $\bar{u} = \bar{s}$ y $\bar{v} = \bar{r}$ como si $\bar{u} = \bar{r}$ y $\bar{v} = \bar{t}$, como se muestra a continuación:

$$\overbrace{\bar{s} \to \cdots\cdots\cdots \to \bar{r}}^{(\bar{u}=\bar{s} \wedge \bar{v}=\bar{r}) \to \psi_{i-1}(\bar{u},\bar{v})} \underbrace{\to \cdots\cdots\cdots \to \bar{t}}_{(\bar{u}=\bar{r} \wedge \bar{v}=\bar{t}) \to \psi_{i-1}(\bar{u},\bar{v})}$$

Llamemos $|\phi|$ al tamaño de la representación de la fórmula ϕ. Notemos que para $i \geq 1$, tenemos que $|\psi_i| \leq |\psi_{i-1}| + O(S(n))$. Para computar ψ_i necesitamos i llamados recursivos hasta llegar al caso base. Luego computamos $\psi_{c \cdot S(n)}$ en tiempo polinomial en $S(n)$, que implica polinomial en n. Notemos que $\psi_{c \cdot S(n)}$ es una QBF generalizada. Usando la Proposición 20, podemos llevarla a forma prenexa en tiempo polinomial. Entonces la equivalencia (22) efectivamente demuestra la reducción polinomial $\mathcal{L} \leq_{\mathrm{p}} \mathsf{CHECKQBF}$ que buscábamos. □

Como consecuencia del Teorema 18 y del Teorema 19, obtenemos

Corolario 3. TQBF $\in$ **PSPACE-completo**.

Teorema 20 (Savitch)**.** **NSPACE**$(S(n)) \subseteq$ **SPACE**$(S(n)^2)$*, de modo que* **PSPACE** = **NPSPACE**.

Demostración. Es un argumento parecido al que usamos para demostrar el Teorema 19. Sea $\mathcal{L} \in$ **NSPACE**$(S(n))$, sea N una máquina no-determinística tal que para todo $x \in \{0,1\}^*$, N acepta x sii $x \in \mathcal{L}$ sii hay un camino de longitud a lo sumo $2^{c \cdot S(|x|)}$ en $G_{N,x}$ desde C_0 hasta

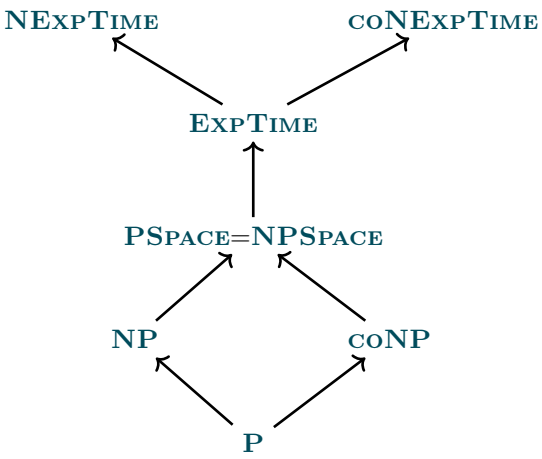


Figura 21: Las clases de complejidad vistas hasta ahora.

C_f. Definimos una función recursiva (traducible a una máquina determinística) *Reach* tal que $Reach(C, C', i) \in \{0, 1\}$ y $Reach(C, C', i) = 1$ si hay un camino de longitud a lo sumo 2^i en $G_{N,x}$ desde C hasta C'. Entonces tenemos que $x \in \mathcal{L}$ sii $Reach(C_0, C_f, c \cdot S(n)) = 1$. $Reach(C, C', i)$ hace esto:

si $i = 0$, devolver 1 si $C \to C'$ en $G_{N,x}$ o 0 si no *(no computa el grafo $G_{N,x}$ entero)*
para cada posible configuración C'' de $G_{N,x}$:
  $j = Reach(C, C'', i-1)$; $k = Reach(C'', C', i-1)$
  si $j = k = 1$, devolver 1
devolver 0

Sea $n = |x|$. $Reach(C, C', i)$ enumera las configuraciones C'' en espacio $O(S(n))$. Reusa el espacio para calcular j y k. Entonces $Reach(C, C', i)$ usa espacio $O(i \cdot S(n))$. Luego, computamos $Reach(C_0, C_f, c \cdot S(n))$ en espacio $O(S(n)^2)$. □

6.3. Espacio logarítmico: L y NL

Cuando analizamos el recurso del tiempo, las funciones computables en tiempo logarítmico no eran muy interesantes porque las máquinas que las computaban no tenían ni tiempo para leer la entrada completa (la entrada tiene tamaño n pero la máquina solo puede usar $O(\log n)$ pasos en el cómputo). Cuando estudiamos el espacio, la situación es diferente, más interesante, porque no contamos las celdas de la entrada. Presentamos dos nuevas clases de complejidad:

Clase de complejidad: L, NL

$$\mathbf{L} = \mathbf{Space}(\log n)$$
$$\mathbf{NL} = \mathbf{NSpace}(\log n)$$

Por un lado, de la definición es inmediato que $\mathbf{L} \subseteq \mathbf{NL}$. Por otro, por la Proposición 18, si S es construible en espacio, sabemos que $\mathbf{NSpace}(S(n)) \subseteq \mathbf{DTime}(2^{O(S(n))})$. Tomando $S(n) = \log n$, concluimos que $\mathbf{NL} \subseteq \mathbf{P}$. Ver Figura 22.

Problema: Cantidad par de 1s

$$\mathsf{EVEN} = \{x \in \{0,1\}^* : \text{hay una cantidad par de 1s en } x\}$$

Ejercicio 11. Probar que $\mathsf{EVEN} \in \mathbf{L}$.

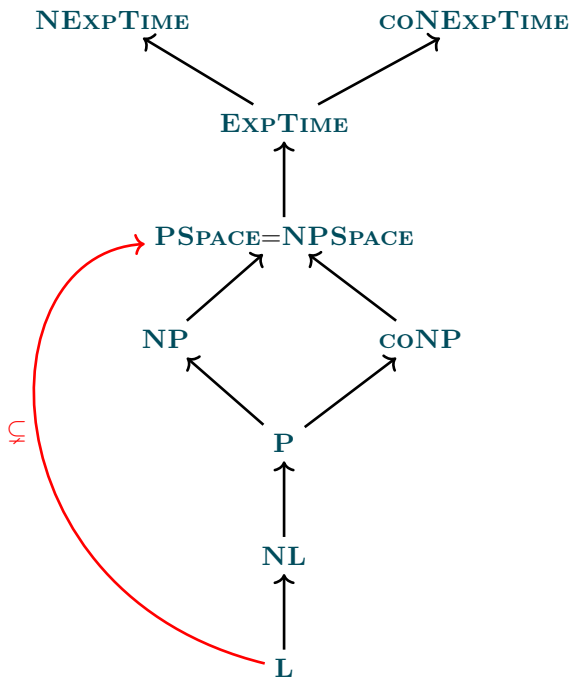


Figura 22: Las clases de complejidad vistas hasta ahora. Por el Teorema 17, $\mathbf{L} \subsetneq \mathbf{PSpace}$.

.

Problema: Existencia de camino

$\mathsf{PATH} = \{\langle G, s, t\rangle$: hay un camino de s a t en el grafo dirigido $G\}$

No se sabe si $\mathsf{PATH} \in \mathbf{L}$, pero

Proposición 23. $\mathsf{PATH} \in \mathbf{NL}$.

Demostración. Suponemos que los nodos de $G = \langle V, E\rangle$ están codificados como números del 0 a $|V| - 1$. Definimos la máquina no-determinística N que con entrada $x = \langle G, s, t\rangle$ hace esto:

$y \leftarrow s$; $m \leftarrow 0$
mientras $m < |V|$:
  $z \leftarrow$ inventar un valor en $\{0, \ldots, |V| - 1\}$
  si $(y, z) \notin E$ (para esto, revisa G en la entrada), pasar a q_{no}
  si no:
    si $z = t$, pasar a $q_{sí}$
    si no:
      $y \leftarrow z$; $m \leftarrow m + 1$
pasar a q_{no}

Este algoritmo inventa no determinísticamente un camino longitud a lo sumo $|V|$. No guarda todo el camino si no que se las arregla para usar solamente tres variables: y, z, m, que se guardan en posiciones contiguas de celdas en la cinta de trabajo. Solo almacenan números menores que $|V|$, de modo que alcanzan $\log |V|$ celdas para almacenar cada variable. Entonces M usa espacio $O(\log n)$, donde $n = |\langle G, s, t\rangle|$. Finalmente, N acepta x sii $x \in \mathsf{PATH}$, y luego $\mathsf{PATH} \in \mathbf{NL}$. □

6.4. Reducibilidad para NL

Para la pregunta $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ usamos la (Karp) reducción polinomial $\leq_{\mathrm{p}}$ vista en la Definición 16. Para la pregunta $\mathbf{L} \stackrel{?}{=} \mathbf{NL}$ no nos sirve $\leq_{\mathrm{p}}$, tal como muestra el siguiente resultado:

Proposición 24. *Si $\mathcal{L} \notin \{\{0,1\}^*, \emptyset\}$ entonces $\mathcal{L}$ es* **NL-hard** *con respecto a $\leq_p$.*

Demostración. Sea $a \in \mathcal{L}$ y $b \notin \mathcal{L}$. Tomemos $\mathcal{L}' \in \mathbf{NL}$. Sea f la función

$$f(x) = \begin{cases} a & \text{si } x \in \mathcal{L}' \\ b & \text{si } x \notin \mathcal{L}' \end{cases}$$

Como $\mathbf{NL} \subseteq \mathbf{P}$, f es computable en tiempo polinomial. Además, $x \in \mathcal{L}'$ sii $f(x) \in \mathcal{L}$. Entonces $\mathcal{L}' \leq_p \mathcal{L}$. □

Trabajaremos con una noción más fuerte de reducción.

Definición 23. Una función f es **computable implícitamente en L** si existe un polinomio p tal que para todo $x \in \{0,1\}^*$, $|f(x)| \leq p(x)$ y tanto $\{\langle x, i\rangle : f(x)(i) = 1\}$ como $\{\langle x, i\rangle : i \leq |f(x)|\}$ están en **L**.

Entonces f es computable implícitamente en **L** cuando la función booleana

$$\langle x, i\rangle \mapsto \begin{cases} f(x)(i) & \text{si } i \leq |f(x)| \\ 0 & \text{si no} \end{cases}$$

está en **L**. Observar que dado $\langle x, i\rangle$, decidir si $i \leq |f(x)|$ también está en **L**.

Definición 24. Una función f es **trabajo-L computable** si existe una máquina determinística que computa f en espacio $O(\log n)$ pero donde solo se cuenta el espacio de las cintas de trabajo y no de la cinta de salida; como siempre, la cinta de salida tiene que ser de solo escritura y cada vez que escribe, se mueve a la derecha.

Ejercicio 12. Probar que una función es computable implícitamente en **L** sii es trabajo-**L** computable.

Proposición 25. *Sean f, g computables implícitamente en* **L**. *Entonces $g \circ f$ es computable implícitamente en* **L**.

Demostración. Sean M_f y M_g máquinas determinísticas que usan espacio $O(\log n)$ tales que $M_f(\langle x, i\rangle) = f(x)(i)$ y $M_g(\langle x, i\rangle) = g(x)(i)$. Definimos la máquina M que con entrada $\langle x, i\rangle$ hace esto:

$j \leftarrow 1$ (índice de la cabeza de entrada ficticia $M_f(\langle x, j\rangle)$)
$k \leftarrow 1$ (índice de la cabeza de salida de $M_f(\langle x, j\rangle)$)
$b \leftarrow$ resultado de simular $M_f(\langle x, j\rangle)$
simular M_g con entrada $M_f(x)$ paso a paso pero con estos cambios:
  engañamos a M_g con la entrada: (el símbolo leído de la entrada de M_g va a ser siempre b)
  sea I la instrucción de M_g a ejecutar
    (depende de b, de las cintas de trabajo de M_g y del estado de M_g)
    si I mueve la cabeza de entrada a R/L,
      incrementa/decrementa j en uno
      $b \leftarrow$ resultado de simular $M_f(\langle x, j\rangle)$
    si I mueve la cabeza de salida (solo puede a R) ,
      incrementa k en uno
    si I escribe $y \in \{0, 1, \square\}$ en la salida y $k = i$,
      escribir y en la cinta de salida de M

Cada vez que M_g lee un bit de $f(x)$, M lo calcula. M no tiene espacio para calcular $f(x)$ entero, porque si bien existe una constante c tal que para todo $x \in \{0,1\}^*$ suficientemente largo tenemos que $|f(x)| \leq |x|^c$, sucede que $|x|^c$ es demasiado grande para ser escrito en una cinta de trabajo.

Para simular $M_f(\langle x, j\rangle)$ necesita leer el x de la entrada original de M porque no tiene espacio para copiarlo en otro lado. M va llevando cuenta de Las cintas de trabajo de M_g, pero además lleva cuenta de las variables j, b y k en cintas de trabajo adicionales. Podemos ver que $M(\langle x, i\rangle) = g(f(x))(i)$. Para las variables j y k usa a lo sumo $\log |f(x)| = \log(|x|^c) = O(\log |x|)$. El resto del espacio que usa es el de que usan M_g y M_f. □

Definición 25. $\mathcal{L}$ es **L-reducible** a $\mathcal{L}'$, notado $\mathcal{L} \leq_\ell \mathcal{L}'$, si existe una función f computable implícitamente en **L** tal que para todo $x \in \{0,1\}^*$, $x \in \mathcal{L}$ sii $f(x) \in \mathcal{L}'$. En este caso decimos que $\mathcal{L} \leq_\ell \mathcal{L}'$ **vía** f.

Clase de complejidad: NL-hard, NL-completo

$\mathcal{L}$ es **NL-hard** si $\mathcal{L}' \leq_\ell \mathcal{L}$ para todo $\mathcal{L}' \in$ **NL**.
$\mathcal{L}$ es **NL-completo** si $\mathcal{L} \in$ **NL** y $\mathcal{L} \in$ **NL-hard**.

Ejercicio 13. Probar que si $\mathcal{L} \leq_\ell \mathcal{L}'$ y $\mathcal{L}' \in$ **L**, entonces $\mathcal{L} \in$ **L**.

Ejercicio 14. Probar que la relación $\leq_\ell$ es transitiva.

6.5. Teorema de Immerman-Szelepcsényi

Teorema 21. PATH $\in$ **NL-completo** *y por lo tanto* $\overline{\text{PATH}} \in$ **coNL-completo**.

Demostración. Sabemos que PATH $\in$ **NL** por la Proposición 23. Veamos que PATH es **NL-hard**. Sea $\mathcal{L} \in$ **NL** y sea N una máquina no-determinística tal que $\mathcal{L}(N) = \mathcal{L}$ y N usa espacio $O(\log n)$. Veamos que $\mathcal{L} \leq_\ell$ PATH vía $f(x) = \langle G_{N,x}, C_0, C_f\rangle$, donde C_0 es la configuración inicial de N para x y C_f es la configuración final. Cada configuración C se codifica con $c \cdot \log |x|$ bits. Tenemos que

$$\begin{array}{lll} x \in \mathcal{L} & \text{sii} & N \text{ acepta } x \\ & \text{sii} & \text{existe un camino desde } C_0 \text{ hasta } C_f \text{ en } G_{N,x} \\ & \text{sii} & f(x) \in \text{PATH}. \end{array}$$

Falta ver que f es computable implícitamente en **L**. El grafo $G_{N,x}$ tiene $2^{c\cdot\log|x|}$ nodos y lo representamos con la matriz de adyacencia, de dimensión $2^{c\cdot\log|x|} \times 2^{c\cdot\log|x|}$. Para esto, antes numeramos todas las configuraciones en orden lexicográfico. Entonces $|f(x)| = O((2^{c\cdot\log|x|})^2) = O(|x|^{2\cdot c})$. Por otro lado, existe una máquina determinística M que dado $\langle x, i\rangle$ decide si i corresponde a un bit de la matriz de adyacencia de $G_{N,x}$, o a un bit de la codificación de C_0 o a un bit de la codificación de C_f. En cada caso, determina si ese bit es 0 o 1 en espacio logarítmico. Por ejemplo, si el bit correspondía a la matriz de adyacencia de $G_{N,x}$ en la posición p, q, la máquina M determina cuál es la configuración C que corresponde a la fila p, cuál es la configuración C' que corresponde a la columna q (recordemos que cada configuración se representa con $O(\log |x|)$ bits) y luego determina si C' es una de las dos posibles evoluciones en un paso desde C revisando la función de transición de N. □

Teorema 22. $\mathcal{L} \in$ **NL** *sii existe un polinomio* $p : \mathbb{N} \to \mathbb{N}$ *y una máquina determinística* M *con una cinta de entrada adicional de lectura de una única vez (lee y pasa a la siguiente celda a la derecha pero no puede volver atrás) tal que 1) para todo* $x \in \{0,1\}^*$, $x \in \mathcal{L}$ *sii existe* $u \in \{0,1\}^{p(|x|)}$ *tal que* $M(x,u) = 1$*; aquí* $M(x,u)$ *denota la salida de* M *cuando la cinta de entrada tiene* x *y la cinta adicional de lectura de una única vez tiene* u*; 2)* M *usa espacio* $O(\log n)$*; en este modelo de máquina con cinta de entrada adicional, la cinta de entrada y la cinta adicional no cuentan en el espacio (solo cuentan sus cintas de trabajo y salida).*

En analogía con la definición de la clase **NP**, en el enunciado anterior, M se llama **verificador** y u se llama **certificado**.

Teorema 23 (Immerman-Szelepcsényi)**.** $\overline{\mathsf{PATH}} \in \mathbf{NL}$.

Demostración. Usaremos la definición de **NL** que surge del Teorema 22. Certificaremos que $\langle G, s, t\rangle \notin \mathsf{PATH}$. Supongamos $G = (V, E)$ y $V = \{1, \ldots, n\}$. Sea

$$A_i = \{v \in V : v \text{ es alcanzable desde } s \text{ en a lo sumo } i \text{ pasos}\}.$$

Notemos que A_n es la componente conexa de s en G. Luego $\langle G, s, t\rangle \notin \mathsf{PATH}$ sii $t \notin A_n$.

En esta demostración vamos a dar varios certificados Z de distintos hechos. Para cada uno habrá un verificador que puede revisar que Z sea un certificado válido para el hecho que pretende certificar. El verificador tiene que seguir estas reglas: 1) solo puede leer Z de izquierda a derecha, de a una celda a la vez y sin volver atrás, 2) solo puede usar espacio logarítmico para su revisión, 3) puede buscar lo que quiera en la cinta de entrada (mover la cabeza a izquierda y derecha), y 4) puede guardar partes de la entrada o partes del certificado en cintas de trabajo, siempre que no excedan el espacio logarítmico. En los certificados, vamos a usar los nodos de G, que se van a representar siembre en binario, de tamaño $O(\log n)$.

Certificado de que $v \in A_i$. Es una lista de nodos

$$Z_{v\in A_i} = \langle v_0, v_1, \ldots, v_k\rangle$$

tal que cada v_i es la codificación en binario de un nodo de V, $v_0 = s$, $(v_j, v_{j+1}) \in E$, $v_k = v$, y $k \leq i$. La lista $Z_{v\in A_i}$ es una forma de probar que $v \in A_i$. Notemos que el tamaño de $Z_{v\in A_i}$ es polinomial y que el verificador puede chequear el certificado en espacio $O(\log n)$.

Certificado de que $v \notin A_i$ conociendo $|A_i|$. Es una lista de pares

$$Z^{|A_i|}_{v\notin A_i} = \langle (v_1, Z_{v_1\in A_i}), (v_2, Z_{v_2\in A_i}), \ldots, (v_k, Z_{v_k\in A_i})\rangle$$

tal que cada v_i es la codificación en binario de un nodo de V, $k = |A_i|$, $v_j < v_{j+1}$, y $v \notin \{v_1, \ldots, v_k\}$. La lista $Z^{|A_i|}_{v\notin A_i}$ muestra k elementos distintos en A_i tal que ninguno es v y $|A_i| = k$. Es una forma de probar que $v \notin A_i$. Notemos que el tamaño de $Z^{|A_i|}_{v\notin A_i}$ es polinomial y que el verificador puede chequear el certificado en espacio $O(\log n)$.

Certificado de que $v \notin A_i$ conociendo $|A_{i-1}|$. Es muy parecido al caso anterior: una lista de pares

$$Z^{|A_{i-1}|}_{v\notin A_i} = \langle (v_1, Z_{v_1\in A_{i-1}}), (v_2, Z_{v_2\in A_{i-1}}), \ldots, (v_k, Z_{v_k\in A_{i-1}})\rangle$$

tal que cada v_i es la codificación en binario de un nodo de V, $k = |A_{i-1}|$, $v_j < v_{j+1}$, $v \notin \{v_1, \ldots, v_k\}$, y $v \notin \bigcup_{1\leq j\leq k} E(v_i)$, donde $E(x) = \{y \in V : (x, y) \in E\}$. Notemos que el tamaño de $Z^{|A_{i-1}|}_{v\notin A_i}$ es polinomial y que el verificador puede chequear el certificado en espacio $O(\log n)$.

Certificado de que $|A_i| = a$ conociendo $|A_{i-1}|$. Es una lista de pares

$$Z^{|A_{i-1}|}_{|A_i|=a} = \langle (1, Z_1), \ldots, (n, Z_n)\rangle$$

tal que: si $v \in A_i$ entonces $Z_v = Z_{v\in A_i}$; si $v \notin A_i$ entonces $Z_v = Z^{|A_i|-1}_{v\notin A_i}$; y $|v : Z_v = Z_{v\in A_i}| = |A_i| = a$. Notemos que el tamaño de $Z^{|A_{i-1}|}_{|A_i|=a}$ es polinomial y que el verificador puede chequear el certificado en espacio $O(\log n)$.

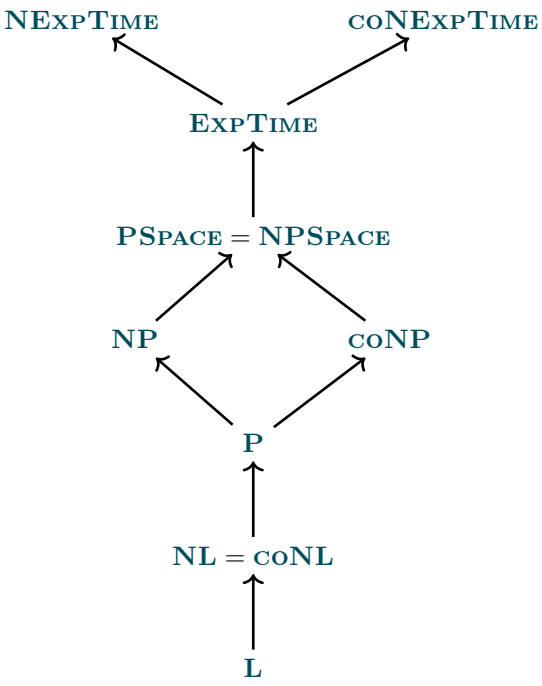


Figura 23: Las clases de complejidad vistas hasta ahora.

Certificado final de que $t \notin A_n$. Es una lista de certificados

$$Z = \langle Z^{|A_0|}_{|A_1|=a_1}, Z^{|A_1|}_{|A_2|=a_2}, \ldots, Z^{|A_{n-1}|}_{|A_n|=a_n}, Z^{|A_n|}_{t \notin A_n} \rangle$$

tal que $a_i = |A_i|$ para todo i. Notemos, una vez más, que el tamaño de Z es polinomial y que el verificador puede chequear el certificado en espacio $O(\log n)$: sabemos que $A_0 = \{s\}$ y por lo tanto $|A_0| = 1$; a medida que lee el certificado de izquierda a derecha, va guardando $|A_{i-1}|$ para verificar $Z^{|A_{i-1}|}_{|A_i|=a_i}$ (reusa el espacio). □

Corolario 4. $\mathbf{NL} = \mathbf{coNL}$.

Demostración. Veamos que $\mathbf{coNL} \subseteq \mathbf{NL}$. Sea $\mathcal{L} \in \mathbf{coNL}$. Por el Teorema 21 sabemos que $\overline{\mathsf{PATH}} \in \mathbf{coNL\text{-}completo}$ y por lo tanto $\mathcal{L} \leq_\ell \overline{\mathsf{PATH}}$. Por el Teorema 23, $\overline{\mathsf{PATH}} \in \mathbf{NL}$ y luego $\mathcal{L} \in \mathbf{NL}$. La prueba para el caso $\mathbf{NL} \subseteq \mathbf{coNL}$ es análoga. □

La relación entre las clases de complejidad vistas hasta ahora se muestra en la Figura 23.

Es importante marcar que acá la situación es distinta a lo que pasa con **NP** y **coNP**, porque $\mathbf{NP} \stackrel{?}{=} \mathbf{coNP}$ es una pregunta abierta, aunque se cree que $\mathbf{NP} \neq \mathbf{coNP}$.

El Teorema 23 puede generalizarse de la siguiente manera:

Teorema 24. *Si $S(n) \geq \log n$ es construible en espacio entonces* $\mathbf{NSpace}(S(n)) = \mathbf{coNSpace}(S(n))$.

Observemos que $\mathbf{NL} = \mathbf{coNL}$ es un caso particular del Teorema 24 cuando $S(n) = \log n$.

7. La jerarquía polinomial

Recordemos el problema SAT de §4.1: para una fórmula booleana φ en CNF, tenemos que $\langle \varphi \rangle \in \mathsf{SAT}$ sii $\exists v\ v \models \varphi$. La propiedad de ser satisfacible empieza con un cuantificador existencial ($\exists$). Si φ tiene n variables y $v \in \{0,1\}^n$, la propiedad "$v \models \varphi$", que es la propiedad que está adentro

del cuantificador existencial, es decidible en tiempo polinomial, dados v y una codificación de φ. Recordemos ahora el problema INDSET. Tenemos que

$$\langle (V,E),k\rangle \in \mathsf{INDSET} \qquad \text{sii} \qquad \exists C\ \ (C \subseteq V \wedge |C| = k \wedge \neg\exists u,v \in C \wedge (u,v) \in E)\,.$$

Aquí, la propiedad de INDSET también empieza con un cuantificador existencial ($\exists$) y la propiedad "$C \subseteq V \wedge |C| = k \wedge \neg\exists u,v \in C \wedge (u,v) \in E$" es decidible en tiempo polinomial, dadas codificaciones del grafo (V,E), del conjunto finito C y de los nodos u,v. Entonces, estos dos problemas se pueden expresar en lógica de primer orden, con una propiedad que empieza con un cuantificador existencial y sigue con una propiedad decidible en tiempo polinomial. Veamos un ejemplo más: el problema TAUT. Es claro que $\langle\varphi\rangle \in \mathsf{TAUT}$ sii $\forall v\ v \models \varphi$. Acá pasa algo parecido a SAT, pero la fórmula que describe al problema empieza con un cuantificador universal ($\forall$) en vez de existencial.

Consideremos ahora el siguiente problema

Problema: Conjunto independiente máximo

$$\mathsf{MAXINDSET} = \{\langle G,k\rangle : \text{el conjunto independiente más grande de } G \text{ tiene } k \text{ vértices}\}$$

Una forma de expresarlo en el lenguaje lógico sería

$$\langle (V,E),k\rangle \in \mathsf{MAXINDSET} \qquad \text{sii} \qquad \exists C\,\forall D \left(\begin{array}{c} C \subseteq V, |C| = k \wedge \neg\exists u,v \in C \wedge (u,v) \in E \wedge \\ (D \subseteq V \wedge \neg\exists u,v \in D \wedge (u,v) \in E) \rightarrow |D| \leq |C| \end{array}\right).$$

Observemos que el predicado entre paréntesis expresa "C es un conjunto independiente de $G = (V,E)$ de k vértices y si D es un conjunto independiente de G, no puede ser más grande que C". Dadas codificaciones del grafo G, de los conjuntos C, D y del número k, decidir si el predicado entre paréntesis vale o no es computable en tiempo polinomial. En este caso, y a diferencia de los ejemplos vistos antes, no parece posible expresar MAXINDSET con un predicado que tenga solo un $\exists$ o solo un $\forall$. Tampoco parece ser posible expresarlo con una fórmula que empiece con un $\forall$. Parecería que la única posibilidad es empezar con un $\exists$ y seguir con un $\forall$.

La jerarquía polinomial clasifica a los problemas según la forma en que pueden ser expresados en lógica. Se usa lógica de primer orden, es decir, se pueden cuantificar variables, que deben contener como valores cadenas binarias de tamaño polinomial. La fórmula que describe al problema será un bloque de cuantificadores seguido de un predicado computable en tiempo polinomial y usará las variables cuantificadas y las variables libres.

Clase de complejidad: $\mathbf{\Sigma_i^p}$, $\mathbf{\Pi_i^p}$

- Para $i > 0$, $\mathbf{\Sigma_i^p}$ es la clase de lenguajes $\mathcal{L}$ tales que existe una máquina determinística M que corre en tiempo polinomial y un polinomio q tal que para todo $x \in \{0,1\}^*$, $x \in \mathcal{L}$ sii

$$\underbrace{\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \ldots Q_i u_i \in \{0,1\}^{q(|x|)}}_{i-1 \text{ alternancias empezando con } \exists}\ M(\langle x,u_1,\ldots,u_i\rangle) = 1$$

  donde $Q_i = \forall$ si i es par y $Q_i = \exists$ en caso contrario.
- $\mathbf{\Sigma_0^p} = \mathbf{P}$
- $\mathbf{PH} = \bigcup_{i\geq 0} \mathbf{\Sigma_i^p}$ es la **jerarquía polinomial**
- Para $i \geq 0$, $\mathbf{\Pi_i^p} = \{\overline{\mathcal{L}} : \mathcal{L} \in \mathbf{\Sigma_i^p}\}$

Se desprende de la definición que para $i > 0$, $\mathbf{\Pi_i^p}$ es la clase de lenguajes $\mathcal{L}$ tales que existe una máquina determinística M que corre en tiempo polinomial y un polinomio q tal que $x \in \mathcal{L}$ sii

$$\underbrace{\forall u_1 \in \{0,1\}^{q(|x|)} \exists u_2 \in \{0,1\}^{q(|x|)} \dots Q_i u_i \in \{0,1\}^{q(|x|)}}_{i-1 \text{ alternancias empezando con } \forall} \; M(\langle x, u_1, \dots, u_i \rangle) = 1$$

donde $Q_i = \exists$ si i es par y $Q_i = \forall$ en caso contrario. Observemos que $\mathbf{NP} = \mathbf{\Sigma_1^p}$ y $\mathbf{coNP} = \mathbf{\Pi_1^p}$.

Lo que importa en $\mathbf{\Sigma_i^p}$ [resp. $\mathbf{\Pi_i^p}$] es que la fórmula empiece con $\exists$ [resp. $\forall$] y tenga $i-1$ *alternancias* de cuantificadores. Se cuentan solo alternancias y no bloques arbitrarios porque bloques $\exists\exists$ se pueden compactar en un solo $\exists$ y, del mismo modo bloques $\forall\forall$ se pueden compactar en un solo $\forall$, como muestra el siguiente ejemplo.

Ejemplo 18. La fórmula

$$\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \exists u_3 \in \{0,1\}^{q(|x|)} \exists u_4 \in \{0,1\}^{q'(|x|)} \quad M(\langle x, u_1, u_2, u_3, u_4 \rangle) = 1$$

es equivalente a

$$\exists u_1 \in \{0,1\}^{q(|x|)+q'(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)+q'(|x|)} \exists u_3' \in \{0,1\}^{q(|x|)+q'(|x|)} \quad M'(\langle x, u_1, u_2, u_3' \rangle) = 1,$$

donde M' solo lee los primeros $q(|x|)$ bits de u_1, u_2 y el resto los ignora. Entonces la información de u_3 y u_4 ahora la tiene pegada en $u_3' = u_3 u_4$ (la primera parte, de longitud $q(|x|)$, corresponde a u_3 y el resto, de longitud $q'(|x|)$, corresponde a u_4.

Hay dos cuestiones que no son esenciales en la definición de $\mathbf{\Sigma_i^p}$. La primera es que podríamos pedir $M(x u_1 \dots u_i) = 1$ en vez de $M(\langle x, u_1, \dots, u_i \rangle) = 1$. Efectivamente, si $M(\langle x, u_1, \dots, u_i \rangle)$ hace el trabajo, hay una máquina determinística M' que dado $y = x u_1 \dots, u_i$ primero calcula n tal que $|y| = n + i \cdot q(n)$. Este n es justamente la longitud de x, de modo que M' sabe qué porción de y representa x y de ahí en adelante conoce también $u_1, \dots, u_i$, porque cada uno tiene longitud $q(n)$. Luego, M' simula M con la entrada $\langle x, u_1, \dots, u_i \rangle$. Si M corre en tiempo polinomial, M' también, así que la definición de $\mathbf{\Sigma_i^p}$ no cambia.

La segunda cuestión es que podríamos pedir distintos polinomios $q_1, \dots, q_i$ para las longitudes de $u_1 \dots, u_i$. Efectivamente, si una máquina determinística M hace el trabajo para $u_1 \dots, u_i$ de tamaños $q_1(|x|), \dots, q_i(|x|)$ respectivamente, podríamos elegir un polinomio q mayor que todos los q_j (por ejemplo, la suma de todos ellos) y usar la definición de $\mathbf{\Sigma_i^p}$ para q y una máquina M' que solo lee los primeros $q_j(|x|)$ bits de u_j para $j = 1, \dots, i$. Si M corre en tiempo polinomial, M' también, así que la definición de $\mathbf{\Sigma_i^p}$ no cambia.

Proposición 26. MAXINDSET $\in \mathbf{\Sigma_2^p}$.

Demostración. Existe una máquina determinística M tal que con entrada $x = \langle V, E, k, C, D \rangle$ ((V, E) representa un grafo G, las variables C y D representan subconjuntos de V y k es un número) hace esto

> *Devolver 1 si C es un conjunto independiente de G de k vértices y, si D es un conjunto independiente de G, el tamaño de C es mayor o igual al tamaño de D; si no, devolver 0.*

Como siempre, suponemos que $V = \{0, \dots, n-1\}$. Los conjuntos C y D los codificamos con palabras en $\{0,1\}^{|V|}$. Como $|V| \leq |x|$, podemos codificar C, D como palabras de $\{0,1\}^{|x|}$, ignorando los bits que no nos interesan. Es claro M corre en tiempo polinomial, porque chequear que C es un conjunto independiente es polinomial, que tiene k vértices también y que si D es un conjunto independiente de G, el tamaño de C es mayor o igual al tamaño de D, también. Además, tenemos que

$$\langle \overbrace{(V,E)}^{G}, k \rangle \in \text{MAXINDSET} \quad \text{sii} \quad \exists C \in \{0,1\}^{|x|} \; \forall D \in \{0,1\}^{|x|} \; M(\langle \overbrace{V, E}^{G}, k, C, D \rangle) = 1$$

y esto prueba que MAXINDSET $\in \mathbf{\Sigma_2^p}$. □

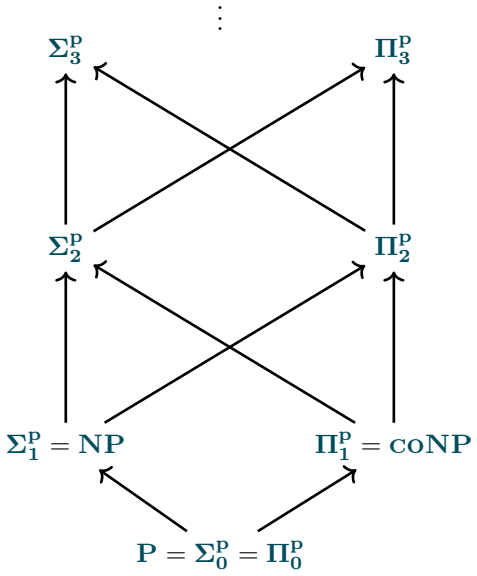


Figura 24: La jerarquía polinomial.

La siguiente proposición enuncia algunas propiedades de la jerarquía polinomial y se representan en la Figura 24. Todas se prueban fácilmente a partir de la definición.

Proposición 27. *La jerarquía polinomial tiene las siguientes propiedades:*

$$\mathbf{\Sigma_1^P = NP},\ \mathbf{\Sigma_i^P \subseteq \Sigma_{i+1}^P},\ \mathbf{\Pi_i^P \subseteq \Sigma_{i+1}^P},\ \mathbf{\Pi_1^P = coNP},\ \mathbf{\Sigma_i^P \subseteq \Pi_{i+1}^P},\ \mathbf{\Pi_i^P \subseteq \Pi_{i+1}^P},\ \mathbf{PH} = \bigcup_{i\geq 0} \mathbf{\Sigma_i^P}$$

La pregunta $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ se puede generalizar a $\mathbf{\Sigma_i^P} \stackrel{?}{=} \mathbf{\Sigma_{i+1}^P}$; ninguna se conoce. Decimos que la jerarquía polinomial **colapsa** si $\mathbf{P} = \mathbf{PH}$. Veamos que si $\mathbf{P} = \mathbf{NP}$ entonces la jerarquía polinomial colapsa.

Proposición 28. *Si* $\mathbf{P} = \mathbf{NP}$ *entonces* $\mathbf{PH} = \mathbf{P}$.

Demostración. Supongamos que $\mathbf{P} = \mathbf{NP}$. Probamos que $\mathbf{\Sigma_i^P}, \mathbf{\Pi_i^P} \subseteq \mathbf{P}$ por inducción en $i \geq 1$. Para $i = 1$ es trivial porque $\mathbf{P} = \mathbf{NP} = \mathbf{\Sigma_1^P}$ y $\mathbf{P} = \mathbf{coNP} = \mathbf{\Pi_1^P}$. Supongamos que $\mathbf{\Sigma_i^P}, \mathbf{\Pi_i^P} \subseteq \mathbf{P}$ y probemos que $\mathbf{\Sigma_{i+1}^P}, \mathbf{\Pi_{i+1}^P} \subseteq \mathbf{P}$. Sea $\mathcal{L} \in \mathbf{\Sigma_{i+1}^P}$. Existe una máquina determinística M que corre en tiempo polinomial y un polinomio q tal que

$$x \in \mathcal{L} \text{ sii } \underbrace{\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \cdots Q_{i+1} u_{i+1} \in \{0,1\}^{q(|x|)}}_{i \text{ alternancias, o sea, } \mathbf{\Sigma_{i+1}^P}} M(\langle x, u_1, \ldots u_{i+1}\rangle) = 1$$

Definimos $\mathcal{L}'$ de esta manera:

$$\langle x, u_1\rangle \in \mathcal{L}' \quad \text{sii} \quad \underbrace{\forall u_2 \in \{0,1\}^{q(|x|)} \cdots Q_{i+1} u_{i+1} \in \{0,1\}^{q(|x|)}}_{i-1 \text{ alternancias, o sea, } \mathbf{\Pi_i^P}} M(\langle x, u_1, \ldots u_{i+1}\rangle) = 1.$$

Como por hipótesis inductiva $\mathcal{L}' \in \mathbf{\Pi_i^P} \subseteq \mathbf{P}$, existe una máquina determinística M' tal que $\mathcal{L}(M') = \mathcal{L}'$ y M' corre en tiempo polinomial. Entonces tenemos que para todo $x \in \{0,1\}^*, u_1 \in \{0,1\}^{q(|x|)}$

$$\langle x, u_1\rangle \in \mathcal{L}' \quad \text{sii} \quad M'(\langle x, u_1\rangle) = 1$$

y esto implica que para todo $x \in \{0,1\}^*$

$$x \in \mathcal{L} \quad \text{sii} \quad \exists u_1 \in \{0,1\}^{q(|x|)}\ M'(\langle x, u_1\rangle) = 1.$$

Luego $\mathcal{L} \in \mathbf{NP} = \mathbf{P}$. Como $\mathcal{L}$ era arbitrario en $\mathbf{\Sigma^p_{i+1}}$, concluimos $\mathbf{\Sigma^p_{i+1}} \subseteq \mathbf{P}$. De la misma manera se prueba $\mathbf{\Pi^p_{i+1}} \subseteq \mathbf{P}$. □

Ejercicio 15. Probar que para todo $i > 0$, si $\mathbf{\Sigma^p_i} = \mathbf{\Pi^p_i}$ entonces $\mathbf{PH} = \mathbf{\Sigma^p_i}$.

Proposición 29. *Las clases $\mathbf{\Sigma^p_i}$ y $\mathbf{\Pi^p_i}$ están cerradas hacia abajo por $\leq_p$. Es decir, para $\mathcal{L}' \leq_p \mathcal{L}$ tenemos que si $\mathcal{L} \in \mathbf{\Sigma^p_i}$ entonces $\mathcal{L}' \in \mathbf{\Sigma^p_i}$ y si $\mathcal{L} \in \mathbf{\Pi^p_i}$ entonces $\mathcal{L}' \in \mathbf{\Pi^p_i}$.*

Demostración. Veamos la demostración para $\mathbf{\Sigma^p_i}$. La demostración para $\mathbf{\Pi^p_i}$ es análoga. Supongamos que $\mathcal{L} \in \mathbf{\Sigma^p_i}$. Entonces existe una máquina determinística M que corre en tiempo polinomial y un polinomio q tal que para todo $x \in \{0,1\}^*$

$$x \in \mathcal{L} \quad \text{sii} \quad \underbrace{\exists u_1 \in \{0,1\}^{q(|x|)} \ldots Q_i u_i \in \{0,1\}^{q(|x|)}}_{i-1 \text{ alternancias}} M(\langle x, u_1, \ldots u_i \rangle) = 1.$$

Supongamos $\mathcal{L}' \leq_p \mathcal{L}$ vía f, es decir f es una función computable en tiempo polinomial tal que para todo $x \in \{0,1\}^*$ tenemos que $x \in \mathcal{L}'$ sii $f(x) \in \mathcal{L}$. Concluimos que

$$x \in \mathcal{L}' \quad \text{sii} \quad \exists u_1 \in \{0,1\}^{q(|x|)} \ldots Q_i \in \{0,1\}^{q(|x|)} \underbrace{M(\langle f(x), u_1, \ldots u_i \rangle)}_{M'(\langle x, u_1, \ldots u_i \rangle)} = 1,$$

donde definimos la máquina M' como $M'(\langle x, u_1, \ldots u_i \rangle) = M(\langle f(x), u_1, \ldots u_i \rangle)$. Como M corre en tiempo polinomial y f es computable en tiempo polinomial, resulta que M' también corre en tiempo polinomial. □

7.1. Problemas $\mathbf{\Sigma^p_i}$-completos

Definimos los mismos conceptos de hardness y completitud para los niveles de la jerarquía polinomial.

Clase de complejidad: $\mathbf{\Sigma^p_i}$-hard, $\mathbf{\Sigma^p_i}$-completo

$\mathcal{L}$ es **$\mathbf{\Sigma^p_i}$-hard** si $\mathcal{L}' \leq_p \mathcal{L}$ para todo $\mathcal{L}' \in \mathbf{\Sigma^p_i}$.
$\mathcal{L}$ es **$\mathbf{\Sigma^p_i}$-completo** si $\mathcal{L} \in \mathbf{\Sigma^p_i}$ y $\mathcal{L} \in \mathbf{\Sigma^p_i}$-hard

Análogamente definimos **$\mathbf{\Pi^p_i}$-hard**, **$\mathbf{\Pi^p_i}$-completo**, **PH-hard**, **PH-completo**.

Se dice que la jerarquía polinomial **colapsa al i-ésimo nivel** si $\mathbf{\Sigma^p_i} = \mathbf{\Sigma^p_{i+1}}$ y en este caso, $\mathbf{\Sigma^p_i} = \mathbf{PH}$. No se sabe si existen los problemas **PH-completos**, pero, de existir uno, la jerarquía polinomial colapsaría en algún nivel.

Proposición 30. *Si existe $\mathcal{L} \in$ **PH-completo** entonces existe i tal que $\mathbf{PH} = \mathbf{\Sigma^p_i}$.*

Demostración. Sea $\mathcal{L} \in \mathbf{PH}$ tal que $\mathcal{L}' \leq_p \mathcal{L}$ para todo $\mathcal{L}' \in \mathbf{PH}$. Como $\mathbf{PH} = \bigcup_i \mathbf{\Sigma^p_i}$, existe i tal que $\mathcal{L} \in \mathbf{\Sigma^p_i}$. Sea $\mathcal{L}' \in \mathbf{PH}$ arbitrario. Por la Proposición 29, como $\mathcal{L}' \leq_p \mathcal{L}$, concluimos $\mathcal{L}' \in \mathbf{\Sigma^p_i}$. Luego $\mathbf{PH} \subseteq \mathbf{\Sigma^p_i}$. Como trivialmente vale $\mathbf{PH} \supseteq \mathbf{\Sigma^p_i}$, concluimos que $\mathbf{PH} = \mathbf{\Sigma^p_i}$. □

Se cree que para todo i, $\mathbf{\Sigma^p_i} \subsetneq \mathbf{\Sigma^p_{i+1}}$, es decir, que la jerarquía polinomial no colapsa al i-ésimo nivel. Por lo tanto, se cree que no existen los problemas **PH-completos**.

Ejercicio 16. Probar que $\mathbf{PH} \subseteq \mathbf{PSpace}$.

Corolario 5. *Si la jerarquía polinomial no colapsa en ningún nivel entonces $\mathbf{PH} \neq \mathbf{PSpace}$.*

Demostración. Probemos el contrarecíproco. Si $\mathbf{PH} = \mathbf{PSpace}$ entonces existirían problemas **PH-completos** por el Corolario 3. Por la Proposición 30, la jerarquía polinomial colapsaría en algún nivel. □

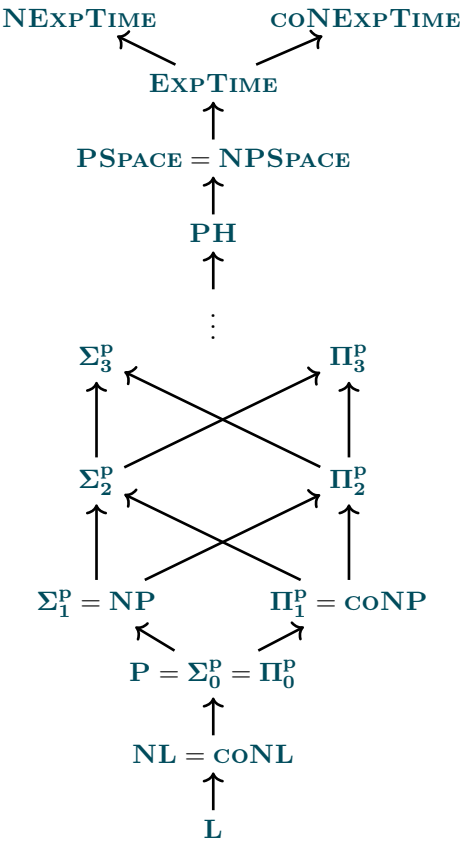


Figura 25: La jerarquía polinomial y su relación con las otras clases de complejidad vistas.

La Figura 25 muestra la jerarquía polinomial y su relación con las clases ya vistas.

Sin pérdida de generalidad podemos extender la definición de QBF dada en (19) a tuplas booleanas. Supongamos que $\bar{x} = x_1, \ldots, x_k$ es una tupla de variables booleanas. Notaremos $|\bar{x}| = k$ a la dimensión de esta tupla.

Problema: Satisfacibilidad de QBF acotada

$\Sigma_i\mathsf{SAT} = \{\langle\varphi\rangle :$ φ es una QBF de la forma $\exists\bar{y}_1\forall\bar{y}_2 \ldots Q_i\bar{y}_i\ \psi(\bar{y}_1, \ldots, \bar{y}_i)$ donde las $\bar{y}_i$ son tuplas de variables booleanas, ψ es una fórmula booleana, los cuantificadores se alternan y $\models \varphi$. $\}$

Observemos que $\Sigma_1\mathsf{SAT} = \mathsf{SAT}$.

Proposición 31. *Para todo $i > 0$, $\Sigma_i\mathsf{SAT} \in \mathbf{\Sigma_i^P}$.*

Demostración. Consideremos la máquina determinística M que con entrada $\langle\varphi, u_1, \ldots, u_i\rangle$ hace esto:

> *si φ no es una QBF de la forma $\exists\bar{y}_1\forall\bar{y}_2 \ldots Q_i\bar{y}_i\ \psi(\bar{y}_1, \ldots, \bar{y}_i)$, o si para algún j, $|u_j| < |\bar{y}_j|$, rechazar. Si no, devolver 1 si $(u_1 \restriction |y_1|) \ldots (u_i \restriction |y_i|) \models \psi$ y 0 en caso contrario*

Es claro que M corre en tiempo polinomial y tenemos que

$$\langle\varphi\rangle \in \Sigma_i\mathsf{SAT} \quad \text{sii} \quad \underbrace{\exists\bar{y}_1 \in \{0,1\}^k \forall\bar{y}_2 \{0,1\}^k \ldots Q_i\bar{y}_i \in \{0,1\}^k}_{i-1 \text{ alternancias}}\ M(\langle\varphi, u_1, \ldots, u_i\rangle)$$

donde $k = |\langle\varphi\rangle|$. Notemos de (21) que la cantidad de variables de φ es siempre menor que la longitud de la codificación de φ (este hecho no es algo particular de la codificación de fórmulas que elegimos; cualquier codificación razonable va a tener la misma propiedad). Observemos también que si x no tiene la forma de una QBF entonces $x \notin \Sigma_i\mathsf{SAT}$ y $M(\langle x, u_1, \ldots, u_i\rangle)$ rechazará $\langle\varphi, u_1, \ldots, u_i\rangle$ independientemente de los valores de $\bar{u}_1, \ldots, \bar{u}_i$. □

Proposición 32. *Para todo $i > 0$, $\Sigma_i\mathsf{SAT} \in \mathbf{\Sigma_i^P}$-hard.*

Demostración. Recordemos la demostración del Teorema 11. De (10) y (11) sabemos que para todo $\mathcal{L} \in \mathbf{NP}$ existe una máquina determinística M que corre en tiempo polinomial t, un polinomio p y una fórmula booleana φ_x tal que para todo $x \in \{0,1\}^*$,

$$x \in \mathcal{L} \quad \text{sii} \quad \exists u \in \{0,1\}^{p(|x|)}\ M(xu) = 1 \quad \text{sii} \quad \varphi_x \in \mathsf{SAT}.$$

La fórmula booleana $\varphi_x = \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4$ era computable en tiempo polinomial a partir de x, una vez que fijábamos M. Además, recordemos de (13) que cada ψ_j ($j = 1 \ldots 4$) tenía variables $p_0, \ldots, p_{2n+3}$ ($n = |x| + p(|x|)$) para codificar la entrada de M (parte corresponde a x y parte al certificado u; se usaban dos variables para codificar cada uno de los símbolos $\{0, 1, \triangleright, \square\}$ que pueden aparecer en la cinta de entrada), y variables $q_1^j, \ldots q_k^j$ (donde k dependía solo de M) para $j = 0, \ldots, m = t(n)$ para codificar cada una de las mini-configuraciones $z_0, \ldots, z_m$ del cómputo de M con entrada xu. Llamemos $\bar{e}$ a la tupla de variables booleanas $p_0, \ldots, p_{2|x|+1}, p_{2n+2}, p_{2n+3}$, donde $p_0, \ldots, p_{2|x|+1}$ codifica la porción inicial de la cinta de entrada conteniendo $\triangleright x$, y p_{2n+2}, p_{2n+3} codifica el símbolo $\square$ después de $\triangleright xu$. Llamemos $\bar{c}$ a la tupla de variables booleanas $p_{2|x|+2}, \ldots, p_{2n+1}$, que codifica porción de la cinta de entrada que corresponde al certificado u. Por último, para cada $j = 1, \ldots, m$, llamemos $\bar{q}_j$ a la tupla de variables booleanas $q_1^j, \ldots q_k^j$, que codifica la mini-configuración z_j. Todas estas tuplas de variables booleanas tienen una dimensión polinomial en $|x|$. En concreto, ψ_1 expresaba que la entrada empezaba con x (fija el valor de las variables de $\bar{e}$ y menciona a las de $\bar{c}$ solo para codificar que $u \in \{0,1\}^*$), ψ_2 expresaba que z_0 es la configuración inicial, ψ_3 expresaba que z_j evolucionaba en un paso en z_{j+1} para $j = 0, \ldots, m-1$, y ψ_4 expresaba que z_m era una configuración final aceptadora de M con entrada x. Lo importante es que

$$\begin{aligned} M(xu) = 1 \quad &\text{sii} \quad \tilde{u} \models \exists\bar{e}, \bar{q}_0, \ldots, \bar{q}_m\ \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \\ &\text{sii} \quad \tilde{u} \models \forall\bar{e}, \bar{q}_0, \ldots, \bar{q}_m\ (\psi_1 \wedge \psi_2 \wedge \psi_3) \rightarrow \psi_4, \end{aligned}$$

donde $|u| = q(|x|)$ y $\tilde{u} = u(0)u(0)u(1)u(1)\dots u(|u|-1)u(|u|-1)$ es la codificación de u y corresponde a la valuación para las variables de $\bar{c}$, las únicas variables libres de las fórmulas booleanas de arriba, que corresponden al certificado de M.

Esto fue solo un repaso de la demostración del Teorema 11. Veamos cómo usarlo ahora para probar la Proposición que estamos queriendo probar. Supongamos que $\mathcal{L} \in \mathbf{\Sigma_i^p}$. Sea M la máquina determinística oblivious, sin cinta de salida y con única cinta de trabajo (recordar la Proposición 3 y la Proposición 5), que corre en tiempo polinomial $t(n)$ y tal que para todo $x \in \{0,1\}^*$ tenemos que

$$x \in \mathcal{L} \quad \text{sii} \quad \underbrace{\exists u_1 \in \{0,1\}^{q(|x|)} \dots Q_i u_i \in \{0,1\}^{q(|x|)}}_{i-1 \text{ alternancias}} \ M(xu_1 \dots u_i) = 1.$$

Si $Q_i = \exists$ tenemos que $x \in \mathcal{L}$ sii $\models \rho_x$ sii $\rho_x \in \Sigma_i\mathsf{SAT}$, donde

$$\rho_x = \underbrace{\exists \bar{c}_1 \forall \bar{c}_2 \dots \overbrace{\exists \bar{c}_i}^{\exists} \exists \bar{e}, \bar{q}_0, \dots, \bar{q}_m}_{i-1 \text{ alternancias}} \ \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4.$$

Observemos que ρ_x es una QBF con todas las tuplas cuantificadas de tamaño polinomial en $|x|$ que se computa en tiempo polinomial a partir de x. Cada $\bar{c}_j$ es una tupla de variables booleanas de dimensión $2 \cdot q(|x|)$ y por lo tanto la tupla $\bar{c}$ descripta más arriba es $\bar{c}_1 \dots \bar{c}_i$, de dimensión $2 \cdot i \cdot q(|x|)$. Observemos también que el último bloque se convierte en un $\exists$, entonces ρ_x sigue teniendo $i-1$ alternancias de cuantificadores. Si $Q_i = \forall$ tenemos que $x \in \mathcal{L}$ sii $\models \rho_x$ sii $\rho_x \in \Sigma_i\mathsf{SAT}$, donde

$$\rho_x = \underbrace{\exists \bar{c}_1 \forall \bar{c}_2 \dots \overbrace{\forall \bar{c}_i}^{\forall} \forall \bar{e}, \bar{q}_0, \dots, \bar{q}_m}_{i-1 \text{ alternancias}} \ (\psi_1 \wedge \psi_2 \wedge \psi_3) \to \psi_4.$$

En este caso, el último bloque se convierte en un $\forall$, entonces ρ_x sigue teniendo $i-1$ alternancias de cuantificadores. Como la función $x \mapsto \rho_x$ es computable en tiempo polinomial, y $x \in \mathcal{L}$ sii $\rho_x \in \Sigma_i\mathsf{SAT}$ concluimos que $\mathcal{L} \leq_\mathrm{p} \Sigma_i\mathsf{SAT}$. Como $\mathcal{L}$ lo tomamos arbitrario en $\mathbf{\Sigma_i^p}$ concluimos que $\Sigma_i\mathsf{SAT}$ es $\mathbf{\Sigma_i^p}$**-hard**. □

De las Proposiciones 31 y 32 concluimos

Corolario 6. *Para todo $i > 0$, $\Sigma_i\mathsf{SAT} \in \mathbf{\Sigma_i^p}$-completo.*

Análogamente,

Problema: Satisfacibilidad de QBF acotada

$\Pi_i\mathsf{SAT} = \{\langle\varphi\rangle :$ φ es una QBF de la forma $\forall \bar{y}_1 \exists \bar{y}_2 \dots Q_i \bar{y}_i \ \psi(\bar{y}_1, \dots, \bar{y}_i)$ donde las $\bar{y}_i$ son tuplas de variables booleanas, ψ es una fórmula booleana, los cuantificadores se alternan y $\models \varphi$. $\}$

Proposición 33. *Para todo $i > 0$, $\Pi_i\mathsf{SAT} \in \mathbf{\Pi_i^p}$-completo.*

7.2. Máquinas con oráculo

Vamos a presentar un nuevo modelo de cómputo, que nos va a servir para entender mejor la jerarquía polinomial. Se llaman **máquinas con oráculo**. Son como las máquinas de Turing que vimos, pero con algunas diferencias. La primera es que su comportamiento depende de un lenguaje $\mathcal{X} \subseteq \{0,1\}^*$, que se llama **oráculo**. La segunda es que, además de las cintas de entrada, la cinta de salida y las cintas de trabajo, tiene una cinta especial que se llama **cinta de consulta**. La tercera diferencia es que tiene tiene tres estados distinguidos más, que se llaman q_{consulta}, $q_{\text{resp:sí}}$, y $q_{\text{resp:no}}$. Finalmente, tiene una nueva instrucción de esta forma:

Si el estado es q_{consulta}, entonces

supongamos que en la cinta de consulta está escrito " $\triangleright\ x\ \square$ ", con $x \in \{0,1\}^*$

- *si $x \in \mathcal{X}$, pasar a $q_{\text{resp:sí}}$*
- *si no, pasar a $q_{\text{resp:no}}$*

Es decir, tiene una instrucción que permite preguntar o consultar al oráculo $\mathcal{X}$ si $x \in \mathcal{X}$. Para hacer esa pregunta, la máquina escribe x en la cinta de consulta y pasa al estado q_{consulta}. La máquina automáticamente (en un solo paso) pasa a $q_{\text{resp:sí}}$ en caso de que $x \in \mathcal{X}$ y pasa a $q_{\text{resp:no}}$ en caso de que $x \notin \mathcal{X}$. Una máquina con oráculo tiene las mismas definiciones de cómputo, aceptación, rechazo, tiempo de cómputo, uso de espacio, etc. que las máquinas que vimos hasta acá, solo que ahora el comportamiento de M depende de la información del oráculo $\mathcal{X}$. Si M es una máquina (determinística o no-determinística) con oráculo, notamos $M^{\mathcal{X}}(x)$ a la salida de M con oráculo $\mathcal{X}$ y entrada x (si es que terminó). Si $M(x)$ termina, solo puede hacer una cantidad finita de consultas al oráculo. Si cambiamos el oráculo en elementos que nunca son consultados, el cómputo no cambia. Entonces, si $M^{\mathcal{X}}(x)$ termina y a lo largo del cómputo consulta $y_1, \dots, y_m$ al oráculo. Para cualquier $\mathcal{Y}$ tal que $y_j \in \mathcal{X}$ sii $y_j \in \mathcal{Y}$ para $j = 1, \dots, m$, tenemos $M^{\mathcal{X}}(x) = M^{\mathcal{Y}}(x)$. Además, $M^{\mathcal{X}}(x)$ al paso t no puede hacer más que t consultas, porque cada instrucción de consulta al oráculo toma un paso en el cómputo, igual que las instrucciones que habíamos visto hasta ahora. Las máquinas con oráculo se pueden codificar de la misma forma que hicimos con las máquinas deterministas o las máquinas no-deterministas. Es importante entender que el oráculo no forma parte de la definición de la máquina sino que es información externa, del mismo modo que la entrada, solo que con una cantidad infinita de información.

Las clases de complejidad que ya estudiamos se pueden relativizar a oráculos:

Clase de complejidad: $\mathbf{P}^{\mathcal{X}}, \mathbf{NP}^{\mathcal{X}}$

- $\mathbf{P}^{\mathcal{X}}$ es la clase de lenguajes decidibles por una máquina determinística que corre en tiempo polinomial y tiene acceso al oráculo $\mathcal{X}$.
- $\mathbf{NP}^{\mathcal{X}}$ es la clase de lenguajes decidibles por una máquina no-determinística que corre en tiempo polinomial y tiene acceso al oráculo $\mathcal{X}$.

Ejemplo 19. Considerar la siguiente máquina determinística M con acceso a SAT y entrada x:

preguntar al oráculo si $x \in$ SAT (escribir x en la cinta de consulta)
si responde 'sí' (entra a $q_{\text{resp:sí}}$), devolver 0
si no (entra a $q_{\text{resp:no}}$), devolver 1

M corre en tiempo lineal independientemente del oráculo al que tenga acceso y $\mathcal{L}(M^{\mathsf{SAT}}) = \overline{\mathsf{SAT}}$.

Proposición 34. *Si $\mathcal{X} \in \mathbf{P}$, entonces $\mathbf{P} = \mathbf{P}^{\mathcal{X}}$.*

Demostración. Probemos que $\mathbf{P}^{\mathcal{X}} \subseteq \mathbf{P}$ (la otra inclusión es trivial). Sea M una máquina determinística que corre en tiempo polinomial y decide $\mathcal{X}$. Sea $\mathcal{L} \in \mathbf{P}^{\mathcal{X}}$ y supongamos que M' es una máquina determinística que corre en tiempo polinomial y decide $\mathcal{L}$ con acceso al oráculo $\mathcal{X}$. Definamos la máquina M'' así: M'' simula M' paso a paso pero reemplaza cada pregunta x al oráculo $\mathcal{X}$ por la llamada a $M(x)$. Entonces, M'' corre en tiempo polinomial y decide $\mathcal{L}$, de modo que $\mathcal{L} \in \mathbf{P}$. □

Problema: Cómputos exponenciales

$$\mathsf{EXPCOM} = \{\langle M, x, 1^n\rangle : \begin{array}{l}\text{la máquina determinística } M \text{ con entrada } x \text{ devuelve} \\ 1 \text{ en a lo sumo } 2^n \text{ pasos}\end{array}\}$$

Proposición 35. $\mathbf{ExpTime} \subseteq \mathbf{P}^{\mathsf{EXPCOM}}$.

Demostración. Sea $\mathcal{L} \in \mathbf{DTime}(2^{n^c})$ para alguna constante c y supongamos que $\mathcal{L} = \mathcal{L}(M)$ para una máquina determinística M que corre en tiempo $O(2^{n^c})$. Supongamos que M corre en tiempo $d \cdot 2^{n^c}$ para alguna constante d. Entonces para todo $x \in \{0,1\}^*$, tenemos que M con entrada x termina en a lo sumo $d \cdot 2^{|x|^c}$ pasos. Luego existe k tal que para todo $x \in \{0,1\}^*$ con $|x| > k$ tenemos que M con entrada x termina en a lo sumo $2^{|x|^{c+1}}$ pasos. Consideremos la siguiente máquina determinística M_1 con entrada x y acceso al oráculo EXPCOM:

si $|x| \leq k$, devolver 1 si $x \in \mathcal{L}$ y 0 en caso contrario
si no, preguntar al oráculo si $\langle M, x, 1^{|x|^{c+1}} \rangle$ y devolver su respuesta

M_1 corre en tiempo polinomial y necesita el acceso al oráculo EXPCOM, entonces $\mathcal{L}(M_1^{\text{EXPCOM}}) \in \mathbf{P}^{\text{EXPCOM}}$. Luego, para todo $x \in \{0,1\}^*$ tenemos $x \in \mathcal{L}(M_1^{\text{EXPCOM}})$ sii $x \in \mathcal{L} = \mathcal{L}(M)$. Entonces M_1 decide $\mathcal{L}$ y por lo tanto $\mathcal{L} \in \mathbf{P}^{\text{EXPCOM}}$. □

Proposición 36. $\mathbf{NP}^{\text{EXPCOM}} \subseteq \mathbf{ExpTime}$.

Demostración. Sea $\mathcal{L} \in \mathbf{NP}^{\text{EXPCOM}}$ y sea N una máquina no-determinística que corre en tiempo polinomial y decide $\mathcal{L}$ con acceso al oráculo EXPCOM. En tiempo exponencial en $|x|$ podemos simular determinísticamente a N con entrada x y también a cada consulta que hace N al oráculo EXPCOM. Luego $\mathcal{L} \in \mathbf{ExpTime}$. □

Corolario 7. $\mathbf{P}^{\text{EXPCOM}} = \mathbf{NP}^{\text{EXPCOM}}$.

Demostración. Por un lado, de la Proposición 35 sabemos que $\mathbf{ExpTime} \subseteq \mathbf{P}^{\text{EXPCOM}}$. Por otro lado, la inclusión $\mathbf{P}^{\text{EXPCOM}} \subseteq \mathbf{NP}^{\text{EXPCOM}}$ es trivial, porque $\mathbf{P} \subseteq \mathbf{NP}$. Finalmente, de la Proposición 36 sabemos que $\mathbf{NP}^{\text{EXPCOM}} \subseteq \mathbf{ExpTime}$. Entonces

$$\mathbf{ExpTime} \subseteq \mathbf{P}^{\text{EXPCOM}} \subseteq \mathbf{NP}^{\text{EXPCOM}} \subseteq \mathbf{ExpTime}$$

y este sandwich termina con la demostración. □

7.3. Teorema de Baker, Gill, Solovay

Muchos resultados se pueden relativizar a oráculos. El siguiente resultado muestra que la pregunta $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ no se puede relativizar.

Teorema 25 (Baker, Gill, Solovay)**.** *Existen oráculos $\mathcal{A}$ y $\mathcal{B}$ tal que $\mathbf{P}^{\mathcal{A}} = \mathbf{NP}^{\mathcal{A}}$ y $\mathbf{P}^{\mathcal{B}} \neq \mathbf{NP}^{\mathcal{B}}$.*

Demostración. La existencia del oráculo $\mathcal{A}$ surge del Corolario 7: basta con tomar $\mathcal{A} = \text{EXPCOM}$.
Veamos cómo construir un $\mathcal{B}$ tal que $\mathbf{P}^{\mathcal{B}} \neq \mathbf{NP}^{\mathcal{B}}$.

Definición de $\mathcal{U}_{\mathcal{B}}$, dado $\mathcal{B}$. Para cualquier $\mathcal{B} \subseteq \{0,1\}^*$ definimos

$$\mathcal{U}_{\mathcal{B}} = \{1^n : \exists x \in \mathcal{B}, |x| = n\}$$

como el conjunto de las longitudes de palabras en $\mathcal{B}$, codificadas en unario. Veamos que $\mathcal{U}_{\mathcal{B}} \in \mathbf{NP}^{\mathcal{B}}$ para cualquier $\mathcal{B}$. Definimos la máquina no-determinística N que con oráculo $\mathcal{B}$ y entrada y hace esto:

si y no es de la forma 1^n para algún n, rechazar
si no (supongamos que $y = 1^n$),
    inventar x tal que $|x| = n$
    consultar si $x \in \mathcal{B}$ y devolver la respuesta

Para todo n tenemos $1^n \in \mathcal{L}(N^{\mathcal{B}})$ sii $\exists x \in \mathcal{B}, |x| = n$ sii $1^n \in \mathcal{U}_{\mathcal{B}}$ (notemos que para y que no sean de la forma 1^n tenemos $y \notin \mathcal{L}(N^{\mathcal{B}})$, $y \notin \mathcal{U}_{\mathcal{B}}$). A continuación definimos un $\mathcal{B}$ para el cual $\mathcal{U}_{\mathcal{B}} \notin \mathbf{P}^{\mathcal{B}}$.

Propiedades que buscamos de $\mathcal{B}$. Para $i \geq 1$, sea M_i la máquina determinística con oráculo representada por la expansión binaria de $i \in \mathbb{N}$ sin el 1 más significativo, tal que para toda máquina con oráculo M existen infinitos i tal que $M = M_i$.

Definiremos $\mathcal{B} = \bigcup_i \mathcal{B}_i$ en etapas y, al mismo tiempo, definimos una sucesión $(n_i)_{i \in \mathbb{N}}$. Las propiedades de $\mathcal{B}_i$ y de n_i que buscamos son estas:

- $\mathcal{B}_0 = \emptyset$ y $n_0 = 1$
- $\mathcal{B}_i \subseteq \mathcal{B}_{i+1}$ y $n_i < n_{i+1}$
- $x \in \mathcal{B}_i \Rightarrow |x| \leq n_i$ (en particular, cada $\mathcal{B}_i$ es finito)

La idea es diagonalizar y lograr que para cada i, si M_i con oráculo $\mathcal{B}$ corre en tiempo polinomial, entonces toma la decisión equivocada cuando la entrada es 1^{n_i}; es decir, si M_i acepta 1^{n_i} entonces nos aseguramos de que ninguna cadena de longitud n_i esté en $\mathcal{B}$ (y por lo tanto $1^{n_i} \notin \mathcal{U}_\mathcal{B}$), mientras que si M_i rechaza 1^{n_i}, entonces metemos en $\mathcal{B}$ alguna cadena de longitud n_i (y por lo tanto $1^{n_i} \in \mathcal{U}_\mathcal{B}$). Nos aseguraremos de que ninguna máquina que corra en tiempo polinomial decide $\mathcal{U}_\mathcal{B}$, de modo que $\mathcal{U}_\mathcal{B} \notin \mathbf{P}^\mathcal{B}$. Más específicamente,

- $M_i^{\mathcal{B}_i}(1^{n_i})$ al tiempo $2^{n_i}-1$ es igual a $M_i^{\mathcal{B}}(1^{n_i})$ al tiempo $2^{n_i}-1$. Es decir, si $M_i^{\mathcal{B}_i}$ con entrada 1^{n_i} paró en a lo sumo $2^{n_i}-1$ pasos, entonces $M_i^{\mathcal{B}}$ con entrada 1^{n_i} también paró en a lo sumo $2^{n_i}-1$ pasos, y $M_i^{\mathcal{B}_i}(1^{n_i}) = M_i^{\mathcal{B}}(1^{n_i})$; y si $M_i^{\mathcal{B}_i}$ con entrada 1^{n_i} no paró al paso $2^{n_i}-1$, entonces $M_i^{\mathcal{B}}$ con entrada 1^{n_i} tampoco paró al paso $2^{n_i}-1$.
- $M_i^{\mathcal{B}_i}$ acepta 1^{n_i} al tiempo $2^{n_i}-1$ sii $\mathcal{B}_i$ no contiene cadenas de tamaño n_i sii $1^{n_i} \notin \mathcal{U}_{\mathcal{B}_i}$ (sii $1^{n_i} \notin \mathcal{U}_\mathcal{B}$, como veremos)

Construcción de $\mathcal{B}_i$ y n_i para $i > 0$. Supongamos que ya tenemos definido n_k para todo $k < i$ y también tenemos definido $\mathcal{B}_{i-1}$. Sea ℓ el máximo de las longitudes consultadas por $M_k^{\mathcal{B}_{i-1}}(1^{n_k})$ al tiempo $2^{n_k}-1$ para todo $k < i$. Definimos $n_i = 1 + \max(\ell, n_{i-1})$. Para definir $\mathcal{B}_i$, simulamos M_i con entrada 1^{n_i} por $2^{n_i}-1$ pasos.

- si M_i consulta al oráculo por un x con $|x| < n_i$, le respondemos lo mismo que '¿$x \in \mathcal{B}_{i-1}$?' Es decir, la hacemos pasar a $q_{\text{resp:sí}}$ si $x \in \mathcal{B}_{i-1}$ y a $q_{\text{resp:no}}$ si $x \notin \mathcal{B}_{i-1}$
- si M_i consulta al oráculo por un x con $|x| \geq n_i$, le respondemos 'no', es decir, la hacemos pasar a $q_{\text{resp:no}}$

Si M_i acepta 1^{n_i} en a lo sumo $2^{n_i}-1$ pasos, definimos $\mathcal{B}_i = \mathcal{B}_{i-1}$. En este caso $\mathcal{B}_i$ no contiene cadenas de longitud n_i, porque todas las cadenas de $\mathcal{B}_{i-1}$ tienen longitud a lo sumo n_{i-1}. Además, $\mathcal{B}$ tampoco va a contener cadenas de longitud n_i, puesto que la única oportunidad que tenía una cadena de longitud n_i de entrar a $\mathcal{B}$ era en el paso i de la construcción. Si M_i rechaza 1^{n_i} o no llegó a una decisión todavía en $2^{n_i}-1$ pasos, elegimos un $x \in \{0,1\}^*$ tal que $|x| = n_i$ que no haya sido consultado y definimos $\mathcal{B}_i = \mathcal{B}_{i-1} \cup \{x\}$. Tal x existe porque simulamos M_i por $2^{n_i}-1$ pasos y por lo tanto hay a lo sumo $2^{n_i}-1$ consultas al oráculo. Sin embargo, hay 2^{n_i} cadenas de tamaño n_i. Observemos que agregar x no 'rompe' ninguna de las simulaciones de M_k a tiempo $2^{n_k}-1$, para $k < i$. A la simulación de M_k por $2^{n_k}-1$ pasos le daba lo mismo el oráculo $\mathcal{B}_k$ o $\mathcal{B}_i$ (o $\mathcal{B}$) porque al paso k, la simulación de M_k no consultaba cadenas de longitud mayor o igual que n_{k+1}. Entonces por inducción en i podemos probar que para todo $k \leq i$, M_k acepta 1^{n_k} sii $\mathcal{B}$ no contiene cadenas de longitud n_k.

Verificación de que $\mathcal{U}_\mathcal{B} \notin \mathbf{P}^\mathcal{B}$. Supongamos que M es una máquina determinística y p es un polinomio tal que $M^\mathcal{B}$ corre en tiempo $p(n)$ y $M^\mathcal{B}$ decide $\mathcal{U}_\mathcal{B}$. Sea i suficientemente grande tal que $M_i = M$ y $2^{n_i}-1 > p(n_i)$. Simular $M_i = M$ por $2^{n_i}-1$ pasos es suficiente para saber si M_i acepta o rechaza 1^{n_i}. Por un lado, si $M_i^\mathcal{B}$ acepta 1^{n_i}, ninguna cadena de longitud n_i está en $\mathcal{B}$; entonces $1^{n_i} \notin \mathcal{U}_\mathcal{B}$. Por otro, si $M_i^\mathcal{B}$ rechaza 1^{n_i}, $\mathcal{B}_i$ contiene una cadena de tamaño n_i; entonces $1^{n_i} \in \mathcal{U}_\mathcal{B}$. Luego $M^\mathcal{B} = M_i^\mathcal{B}$ no puede decidir $\mathcal{U}_\mathcal{B}$ porque falla para la entrada 1^{n_i}. □

7.4. La jerarquía polinomial y NP con oráculos

Definimos las clases de complejidad **P** y **NP** relativas a clases de complejidad de esta forma:

Clase de complejidad: $\mathbf{P^C}, \mathbf{NP^C}$

$$\mathbf{P^C} = \bigcup_{\mathcal{X} \in \mathbf{C}} \mathbf{P}^{\mathcal{X}}$$
$$\mathbf{NP^C} = \bigcup_{\mathcal{X} \in \mathbf{C}} \mathbf{NP}^{\mathcal{X}}$$

Observemos que si $\mathcal{X} \in$ **C-completo**, entonces $\mathbf{P^C} = \mathbf{P}^{\mathcal{X}}$ y $\mathbf{NP^C} = \mathbf{NP}^{\mathcal{X}}$.

Ejemplo 20. $\mathbf{P^{NP}} = \mathbf{P}^{\mathsf{SAT}} = \mathbf{P}^{\overline{\mathsf{SAT}}} = \mathbf{P^{coNP}}$.

El siguiente resultado establece relaciones entre la jerarquía polinomial y las clases **NP** relativas a oráculos.

Teorema 26. *Para* $i \geq 1$, $\mathbf{\Sigma^p_{i+1}} = \mathbf{NP}^{\Sigma_i \mathsf{SAT}}$.

Demostración. Veamos primero que $\mathbf{\Sigma^p_{i+1}} \subseteq \mathbf{NP}^{\Sigma_i \mathsf{SAT}}$. Sea $\mathcal{L} \in \mathbf{\Sigma^p_{i+1}}$. Existe una máquina determinística M que corre en tiempo polinomial y un polinomio q tal que

$$x \in \mathcal{L} \quad \text{sii} \quad \underbrace{\exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \dots Q u_{i+1} \in \{0,1\}^{q(|x|)}}_{i \text{ alternancias}} \; M(\langle x, u_1, \dots, u_i \rangle) = 1.$$

Definamos

$$\mathcal{L}' = \{\langle x, u_1 \rangle : \underbrace{\forall u_2 \{0,1\}^{q(|x|)} \dots Q_{i+1} u_{i+1} \{0,1\}^{q(|x|)}}_{i-1 \text{ alternancias}} \; M(\langle x, u_1, \dots, u_i \rangle) = 1\}.$$

Como $\mathcal{L}' \in \mathbf{\Pi^p_i}$, por el Corolario 6 sabemos que $\mathcal{L}' \leq_{\mathrm{p}} \overline{\Sigma_i \mathsf{SAT}}$. Definimos una máquina no-determinística N que con oráculo $\mathcal{L}'$ y entrada $x \in \{0,1\}^*$ hace esto:

inventar u_1
consultar si $\langle x, u_1 \rangle \in \mathcal{L}'$ *y devolver su respuesta*

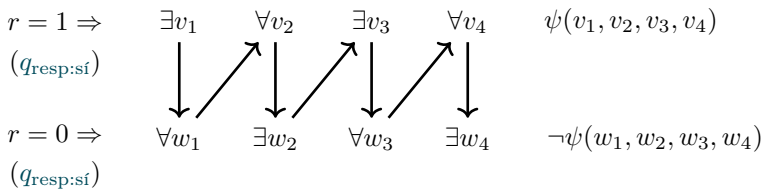
Es claro que N corre en tiempo lineal y que $\mathcal{L}(N^{\mathcal{L}'}) = \mathcal{L}$. Además el oráculo $\mathcal{L}'$ cumple que $\mathcal{L}' \leq_{\mathrm{p}} \overline{\Sigma_i \mathsf{SAT}}$. Entonces $\mathcal{L} \in \mathbf{NP}^{\overline{\Sigma_i \mathsf{SAT}}} = \mathbf{NP}^{\Sigma_i \mathsf{SAT}}$.

Veamos ahora que $\mathbf{NP}^{\Sigma_i \mathsf{SAT}} \subseteq \mathbf{\Sigma^p_{i+1}}$. Sea $\mathcal{L} \in \mathbf{NP}^{\Sigma_i \mathsf{SAT}}$ y sea N una máquina no-determinística que corre en tiempo polinomial t tal que con oráculo $\Sigma_i \mathsf{SAT}$ decide $\mathcal{L}$. Entonces $x \in \mathcal{L}$ sii existe un cómputo $u \in \{0,1\}^{t(|x|)}$ de $N^{\Sigma_i \mathsf{SAT}}$ con entrada x que llega a $q_{\text{sí}}$ (recordemos de §3.3 que los cómputos de máquinas no-determinísticas se pueden codificar con cadenas binarias). A lo largo de este cómputo u, N hace consultas $\varphi_1, \dots, \varphi_k$ $(k \leq t(|x|))$ al oráculo, del tipo

$$\varphi_j = \underbrace{\exists \bar{u}_1^j \forall \bar{u}_2^j \dots Q \bar{u}_i^j}_{i-1 \text{ alternancias}} \; \psi_j(\bar{u}_1, \dots, \bar{u}_i)$$

donde ψ_j $(j = 1, \dots, k)$ es una fórmula booleana y los $\bar{u}_i^j$ son tuplas de variables booleanas, y recibe respuesta $r_j \in \{0,1\}$ ('sí'$= q_{\text{resp:sí}} = 1$ o 'no'$= q_{\text{resp:no}} = 0$)

- si $r_j = 1$ entonces existe $\bar{v}_1^j$ tal que $\bar{v}_1^j \models \forall \bar{v}_2^j \dots Q \bar{v}_i^j \; \psi_j(\bar{v}_1^j, \bar{v}_2^j, \dots, \bar{v}_i^j)$;
- si $r_j = 0$ entonces para todo $\bar{w}_1^j$ tenemos $\bar{w}_1^j \not\models \forall \bar{w}_2^j \dots Q \bar{w}_i^j \; \psi_j(\bar{w}_1^j, \bar{w}_2^j, \dots, \bar{w}_i^j)$, o sea $\bar{w}_1^j \models \exists \bar{w}_2^j \dots \overline{Q} \bar{w}_i^j \; \neg \psi_j(\bar{w}_1^j, \bar{w}_2^j, \dots, \bar{w}_i^j)$, donde $\overline{\exists} = \forall$ y $\overline{\forall} = \exists$.



$$x \in \mathcal{L} \quad \text{sii} \quad \underbrace{\exists u \quad \exists r \quad \exists v_1 \quad \forall w_1 \forall v_2 \ \exists w_2 \exists v_3 \ \forall w_3 \forall v_4 \ \exists w_4}_{\text{4 alternancias, es decir, } \mathbf{\Sigma_5^p}} \quad \text{[Propiedad tiempo polinomial]}$$

Figura 26: Supongamos que N acepta x y a lo largo del computo u, N hace 1 sola consulta r al oráculo Σ_4SAT: $\varphi = \exists x_1 \forall x_2 \exists x_3 \forall x_4 \psi$, con ψ una fórmula booleana.

Entonces $x \in \mathcal{L}$ sii existe un cómputo u, variables booleanas $(r_j)_{j=1,\dots,k}$ y tuplas booleanas $(\bar{v}_1^j)_{j=1,\dots,k}$ y $(\bar{w}_1^j)_{j=1,\dots,k}$ tal que N acepta x siguiendo el cómputo u, recibe como respuestas $r_1, \dots, r_k$ (en orden) a las consultas al oráculo y para $j = 1, \dots, k$

- $r_j = 1$ y $\bar{v}_1^j \models \forall \bar{v}_2^j \exists \bar{v}_3^j \dots Q \bar{v}_i^j \ \psi_j(\bar{v}_1^j, \bar{v}_2^j, \dots \bar{v}_i^j)$ para algún $\bar{v}_1^j$ o bien
- $r_j = 0$ y $\bar{w}_1^j \models \exists \bar{w}_2^j \forall \bar{w}_3^j \dots \overline{Q} \bar{w}_i^j \ \neg\psi_j(\bar{w}_1^j, \bar{w}_2^j, \dots \bar{w}_i^j)$ para todo $\bar{w}_1^j$.

Luego,

$$x \in \mathcal{L} \quad \text{sii} \quad \underbrace{\overbrace{\exists u, (r_j)_j, (\bar{v}_1^j)_j}^{\exists} \overbrace{\forall (\bar{w}_1^j)_j \forall (\bar{v}_2^j)_j}^{\forall} \overbrace{\exists (\bar{w}_2^j)_j \exists (\bar{v}_3^j)_j \forall (\bar{w}_3^j)_j \dots Q(\bar{v}_i^j)_j \overline{Q}(\bar{w}_i^j)_j}^{i-2 \text{ alternancias}}}_{i \text{ alternancias}} \ \varphi,$$

donde φ expresa que N acepta x siguiendo el cómputo u, recibe como respuestas $r_1, \dots, r_k$ (en orden) a las consultas al oráculo y para $j = 1, \dots, k$: $r_j = 1$ y $\models \psi_j(\bar{v}_1^j, \bar{v}_2^j, \dots, \bar{v}_i^j)$ o bien $r_j = 0$ y $\models \neg\psi_j(\bar{w}_1^j, \bar{w}_2^j, \dots, \bar{w}_i^j)$. Ver Figura 26 para un ejemplo. No es difícil ver que φ expresa una condición que es computable en tiempo polinomial. Concluimos así que $\mathcal{L} \in \mathbf{\Sigma_{i+1}^p}$. □

Ejemplo 21. Primero, observemos que $\mathbf{\Sigma_1^p} = \mathbf{NP}$; este caso no cae dentro de las hipótesis del Teorema 26 sino que ya lo sabíamos. Ahora, recordando que Σ_1SAT = SAT, el Teorema 26 nos dice que $\mathbf{\Sigma_2^p} = \mathbf{NP}^{\Sigma_1\text{SAT}} = \mathbf{NP}^{\text{SAT}} = \mathbf{NP}^{\mathbf{NP}}$ y que $\mathbf{\Sigma_3^p} = \mathbf{NP}^{\Sigma_2\text{SAT}} = \mathbf{NP}^{\mathbf{NP}^{\mathbf{NP}}}$.

Terminamos la sección con una definición de complejidad más. La Figura 27 muestra las clases de la jerarquía polinomial y su relación con **NP** relativo a oráculos.

Clase de complejidad: $\mathbf{\Delta_i^p}$

$$\mathbf{\Delta_1^p} = \mathbf{P}$$
$$\mathbf{\Delta_{i+1}^p} = \mathbf{P}^{\mathbf{\Sigma_i^p}}$$

Ejemplo 22. $\mathbf{\Delta_2^p} = \mathbf{P}^{\mathbf{\Sigma_1^p}} = \mathbf{P}^{\mathbf{NP}}$ y $\mathbf{\Delta_3^p} = \mathbf{P}^{\mathbf{\Sigma_2^p}} = \mathbf{P}^{\mathbf{NP}^{\mathbf{NP}}}$.

Ejemplo 23. El siguiente problema es $\mathbf{\Delta_2^p}$-completo.

Problema: LEXSAT **(Mínima valuación en orden lex)**

LEXSAT $= \{\langle v, \varphi \rangle :$ v es la valuación más chica en orden lexicográfico tal que $v \models \varphi\}$

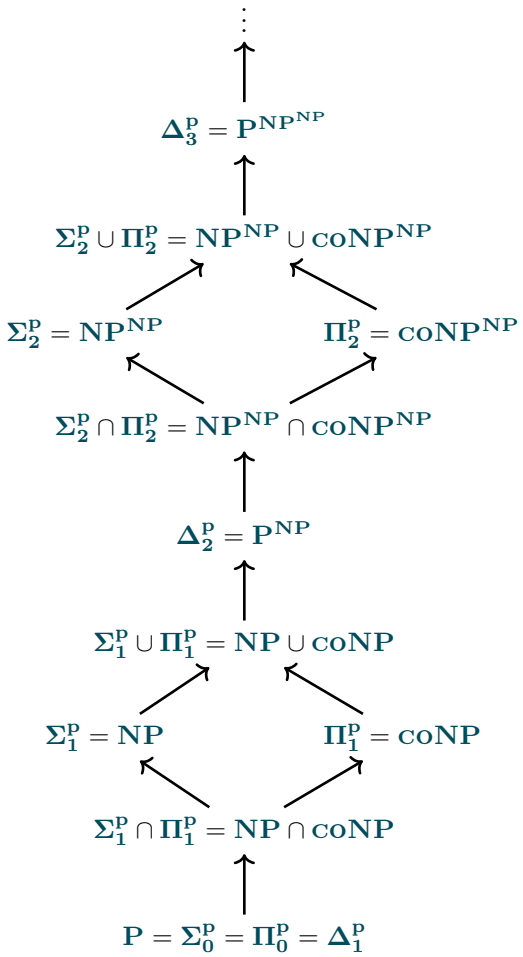


Figura 27: La jerarquía polinomial y **NP** relativo a oráculos.

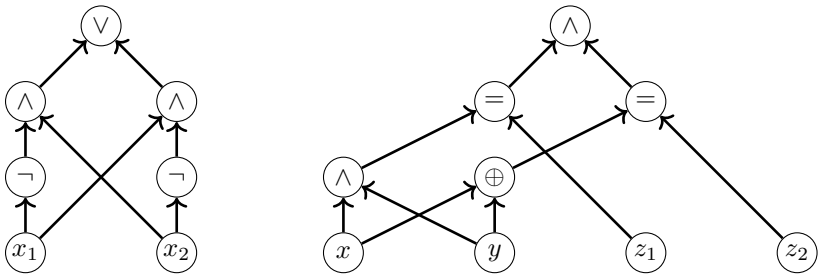
8. Circuitos booleanos

En este capítulo vamos a cambiar el modelo de cómputo. Trabajaremos con los circuitos booleanos.

Definición 26. Un **circuito booleano** (o simplemente **circuito**) de n entradas es un grafo acíclico dirigido tal que tiene n **entradas** (*sources*), que son nodos sin flechas de entrada y tiene 1 nodo destino o **salida** (*sink*), que es un nodo sin flecha de salida. El resto de los nodos se llaman **compuertas** (*gates*) y están etiquetados con

- $\neg$: *fan-in* =1, *fan-out* =1
- $\wedge$: *fan-in* =2, *fan-out* =1
- $\vee$: *fan-in* =2, *fan-out* =1

El **tamaño** $|C|$ de un circuito booleano C es la cantidad de vértices de C.



a. *b.*

Figura 28: *a.* La entrada son los nodos x_1 y x_2; la salida es el nodo con etiqueta $\vee$. $C(x_1, x_2) = x_1 \oplus x_2 = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$, que es la definición del 'o exclusivo'. *b.* La entrada son los nodos x, y, z_1, z_2. El nodo $\oplus$ representa el circuito para el 'o exclusivo' (ver Figura *a.*) y el nodo $=$ representa un circuito para el $\Leftrightarrow$ ('sii' lógico), que es la negación del circuito *a.* $C(x, y, z_1, z_2) = ((x \wedge y) = z_1) \wedge ((x \oplus y) = z_2)$.

La semántica, es decir, el funcionamiento del circuito es el siguiente. Dados valores booleanos para los nodos de entrada, la información fluye en la dirección de las flechas y cada compuerta realiza la operación con la que está etiquetada. La salida es el valor booleano del nodo de salida. La definición formal es la siguiente:

Definición 27. Dado un circuito booleano $C = (V, E)$ con entradas $X = \{x_1, \ldots, x_n\}$ y $x \in \{0,1\}^n$, definimos recursivamente $v : V \to \{0,1\}$ de esta forma:

- $v(x_i) = x(i)$
- si en C tenemos $u_1 \to u$ (u tiene etiqueta $\neg$) y u tiene *fan-in* $=1$ entonces $v(u) = \neg v(u_2)$
- si en C tenemos $u_1 \to u \leftarrow u_2$, y
  - u tiene etiqueta $\wedge$, entonces $v(u) = v(u_1) \wedge v(u_2)$
  - u tiene etiqueta $\vee$, entonces $v(u) = v(u_1) \vee v(u_2)$

Definimos $C(x)$ como $v(t)$, donde t es nodo de salida de C. El circuito C calcula la función $x \mapsto C(x)$.

La Figura 28 muestra dos ejemplos de circuitos booleanos C y la función $C(x)$ que calcula.

8.1. La clase $\mathbf{P}_{/\mathbf{poly}}$

Definición 28. Sea $S : \mathbb{N} \to \mathbb{N}$. Una **familia de circuitos** de **tamaño** $S(n)$ es una secuencia $(C_n)_{n\in\mathbb{N}}$ tal que C_n es un circuito con n entradas y $|C_n| \leq S(n)$.

Definición 29. Una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ [de tamaño $S(n)$] **decide** el lenguaje $\mathcal{L}$ si para todo $n \geq 1$ y todo $x \in \{0,1\}^n$ tenemos que $x \in \mathcal{L}$ sii $C_n(x) = 1$. Como siempre, identificamos la tupla $(x_1, \ldots, x_n) \in \{0,1\}^n$ con la cadena $x_1 \ldots x_n$.

Observemos que según esta definición alcanza con que exista la familia de circuitos $(C_n)_{n\in\mathbb{N}}$. No pedimos que sea **uniforme**, es decir que exista una máquina determinística M tal que $M(n) = \langle C_n \rangle$.

Clase de complejidad: SIZE$(S(n))$

SIZE$(S(n))$ es la clase de lenguajes $\mathcal{L}$ ($\epsilon \notin \mathcal{L}$) tal que existe una familia de circuitos $(C_n)_{n\in\mathbb{N}}$

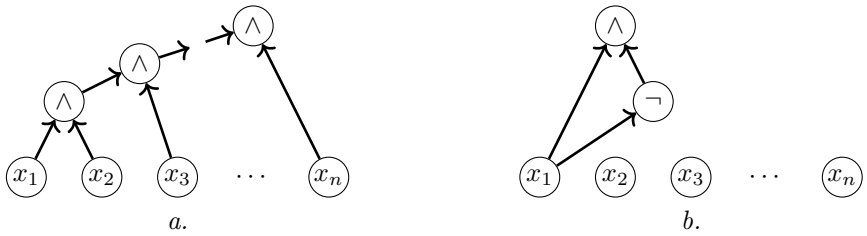


Figura 29: *a.* un circuito C_n tal que $C_n(x) = 1$ sii $x = 1^n$. *b.* un circuito C_n tal que $C_n(x) = 0$ para todo $x \in \{0,1\}^n$

de tamaño $S(n)$ que decide $\mathcal{L}$.

Ejemplo 24. Supongamos que $\mathcal{L}$ es cualquier lenguaje unario, es decir $\mathcal{L} \subseteq \{1^n : \ \geq 1\}$. Si $x = x_1 \dots x_n \in \mathcal{L}$ definimos C_n como en la Figura 29-*a* y si no, definimos C_n como en la Figura 29-*b*.

Ejemplo 25. Veamos cómo sumar en binario. Consideremos el siguiente problema:

Problema: Suma en binario

$$\mathsf{SUMA} = \{xyz : |x| = |y| = i, |z| = i+1, x + y = z, i \geq 1\}$$

Ver la Figura 28-*b*: para $n = 4$ tenemos $0000, 0101, 1001, 1110 \in \mathsf{SUMA}$; el resto de las palabras de largo 4 no están en SUMA. En general, $x_n \dots x_1 \ y_n \dots y_1 \ z_{n+1} \dots z_1 \in \mathsf{SUMA}$ sii existen $c_1 \dots, c_{n+1}$ tal que $c_1 = 0$, para $j = 1 \dots n$:

$$z_j = x_j \oplus y_j \oplus c_j \tag{24}$$

$$c_{j+1} = (c_j \wedge (x_j \vee y_j)) \vee (x_j \wedge y_j)), \tag{25}$$

y $z_{n+1} = c_{n+1}$. Aquí cada c_i es un resultado intermedio que funciona como nodo en el circuito y representa el bit de *carry* en la suma de derecha a izquierda que todos conocemos, $\oplus$ representa el circuito para el 'o exclusivo' (ver Figura 28-*a*) y $=$ representa el circuito para el $\Leftrightarrow$, el 'sii' lógico, que se construye negando la salida del circuito para el $\oplus$. Se puede ver que $\mathsf{SUMA} \in \textsc{Size}(S(n))$ para S lineal.

Veamos cómo representar en binario un circuito C de S nodos. Supongamos que los nodos de C son $u_0, \dots, u_{S-1}$, de modo tal que $u_1, \dots u_n$ corresponden a la entrada y u_0 a la salida. Codificamos cada nodo u con $[u]$, es decir, con $\lceil \log S \rceil$ bits. Cada nodo de C tiene a lo sumo dos hijos (porque las operaciones son unarias o binarias). Codificamos el circuito C con la palabra $\tau_0\sigma_0 \ \dots \ \tau_{S-1}\sigma_{S-1}$, donde

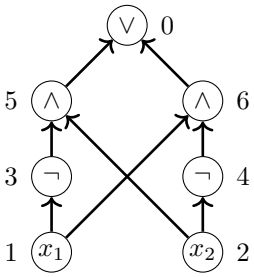
- si u_i es una entrada (y por lo tanto no tiene hijos), $\tau_i = 00$ y $\sigma_i = \varepsilon$ (en realidad alcanza con codificar la cantidad de entradas)
- si u_i tiene etiqueta $\neg$ (tiene solo 1 hijo v_i), $\tau_i = 01$ y $\sigma_i = [v_i]$
- si u_i tiene etiqueta $\wedge$ y tiene hijos v_i, w_i, $\tau_i = 10$ y $\sigma_i = [v_i][w_i]$
- si u_i tiene etiqueta $\vee$ y tiene hijos v_i, w_i, $\tau_i = 11$ y $\sigma_i = [v_i][w_i]$

Así, codificamos C con a lo sumo

$$S \cdot (2 \cdot \lceil \log S \rceil + 2) \leq 4 \cdot S \cdot \log S \tag{26}$$

bits.

Ejemplo 26. La Figura 30 muestra un ejemplo de codificación de un circuito.



nodo u_i	tipo τ_i	hijos σ_i	
0	$\vee$	5	6
1	entrada		
2	entrada		
3	$\neg$	1	
4	$\neg$	2	
5	$\wedge$	3	2
6	$\wedge$	1	4

tipo τ_i	hijos σ_i	
11	101	110
00		
00		
01	001	
01	010	
10	100	010
10	001	010

Figura 30: Un circuito y su codificación en binario, 11 101 110 00 00 01 001 01 010 10 100 010 10 001 010 (los espacios no forman parte de la codificación).

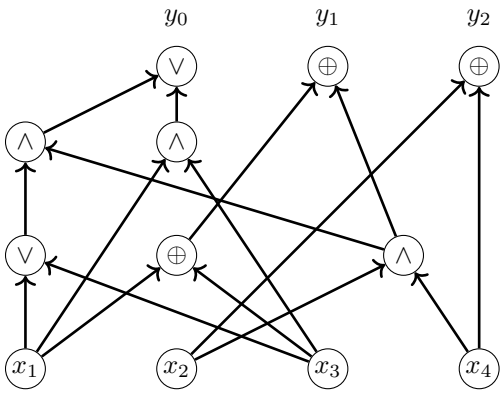


Figura 31: La suma en binario de dos cadenas de longitud 2. La entrada es x_1, x_2, x_3, x_4 y la salida es y_0, y_1, y_2 ($y_i \in \{0,1\}$) tal que $y_0, y_1, y_2 = x_1x_2 + x_3x_4$, es decir, $y_0y_1y_2$ es la representación en binario de la suma del número x_1x_2 y el número x_3x_4. Por ejemplo, $10 + 10 = 100$.

Hasta ahora vimos circuitos con una sola salida. No hay nada especial con esta característica. Todo lo que dijimos sirve para circuitos con m salidas y n entradas, para representar funciones $\{0,1\}^n \to \{0,1\}^m$. Tenemos n nodos de entrada (con *fan-in* $= 0$), m nodos de salida (con *fan-out* $= 0$), las representaciones en binario son análogas a las que vimos solo que ahora representamos todos los nodos de salida.

Ejemplo 27. El circuito de la Figura 31 calcula $x_1x_2x_3x_4 \mapsto y_0y_1y_2$, donde $y_0y_1y_2 = x_1x_2 + x_3x_4$ y el '+' representa la suma en binario.

Clase de complejidad: $\mathbf{P}_{/\mathbf{poly}}$

$$\mathbf{P}_{/\mathbf{poly}} = \bigcup_c \textsc{Size}(c \cdot n^c)$$

Teorema 27. $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$.

Demostración. Recordemos una vez más la demostración del Teorema 11. Sea $\mathcal{L} \in \mathbf{P}$ y sea M una máquina determinística tal que M corre en tiempo $t(n)$, con t un polinomio, y $x \in \mathcal{L}$ sii $M(x) = 1$ para todo $x \in \{0,1\}^*$. Por la Proposición 3, podemos suponer que M es oblivious y por la Proposición 5, podemos suponerla también sin cinta de salida. Como siempre, la suponemos también con una única cinta de trabajo. Entonces, $x \in \mathcal{L}$ sii existe una secuencia de mini-configuraciones $z_0, \ldots, z_{t(|x|)}$ de M con entrada x tal que z_0 es inicial para x, z_i evoluciona en un paso en z_{i+1} para $i = 0, \ldots, t(|x|) - 1$ y $z_{t(|x|)}$ es final. Representamos z_i con la cadena ets, donde:

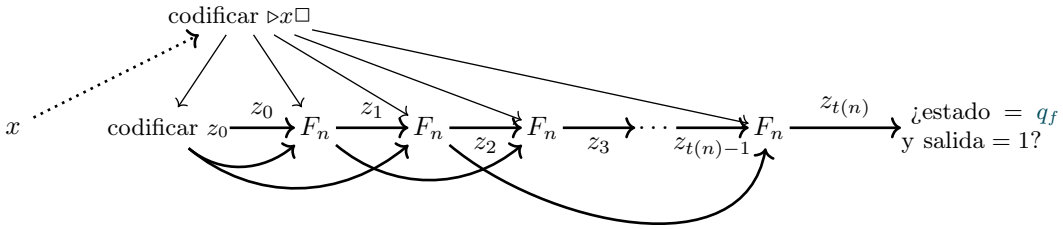


Figura 32: El esquema del circuito C_n de n entradas es tal que $C_n(x) = M(x)$, donde $x \in \{0,1\}^n$, M es una máquina determinística oblivious con una sola cinta de trabajo y sin cinta de salida que corre en tiempo $t(n)$, y donde z_i son las mini-configuraciones del cómputo de M con entrada x y F_n es el circuito de tamaño constante que representa la función $F_n : \{0,1\}^{2k+2} \to \{0,1\}^k$ definida en (27). Las posiciones de las cintas solo dependen de n y del número de paso del cómputo de M con entrada $x \in \{0,1\}^n$. El nodo final de salida representa un circuito de tamaño constante que decide si el estado de $z_{t(n)}$ es q_f y el bit de la cinta de trabajo en $z_{t(n)}$ es 1. El nodo 'codificar $\triangleright x\square$' representa un circuito de tamaño $O(n)$ que modifica la entrada para codificar el símbolo inicial y final de la cinta de entrada; cada símbolo de $\{0, 1, \triangleright, \square\}$ lo codifica con 2 bits. El nodo 'codificar z_0' representa un circuito de tamaño constante que devuelve la codificación de la mini-configuración z_0. Todas las flechas salvo la punteada representan una cantidad constante de ejes dirigidos del circuito; las flechas más finas representan solo 2 ejes dirigidos y corresponden a la codificación del símbolo leído en la cinta de entrada por M en cada paso del cómputo. Las flechas gruesas corresponden a k ejes dirigidos. Para $i > 0$, cada circuito F_n calcula un z_i y tiene $2k + 2$ ejes dirigidos de entrada: k corresponden a la entrada z_{i-1} (flecha recta), k corresponden a la entrada $z_{prev(i,n)}$ (flecha arqueada) y 2 corresponden a la codificación del símbolo leído por la cabeza de entrada (flecha fina).

- $e \in \{0,1\}^2$ codifica el símbolo leído por la cabeza en la cinta de entrada,
- $t \in \{0,1\}^2$ codifica el símbolo leído por la cabeza en la cinta de trabajo,
- $s \in \{0,1\}^c$ codifica el estado de M (c es constante)

Entonces el tamaño de la codificación de cada z_i es constante igual a $4 + c = k$.

Para n fijo, construimos un circuito C_n tal que $M(x) = C_n(x)$ para todo $x \in \{0,1\}^n$. Como vimos, z_{i+1} solo depende del símbolo leído por la cabeza de entrada (codificado con 2 bits), la mini-configuración z_{i-1} y la mini-configuración $z_{prev(i,n)}$, donde $prev(i,n)$ es el máximo paso $j < i$ en el cómputo de M con entrada 0^n tal que la posición de la cabeza de trabajo en el paso j coincide con la posición de la cinta de trabajo en el paso i. Sea $F_n : \{0,1\}^{2k+2} \to \{0,1\}^k$ tal que

$$F_n(e\ z_i\ z_{prev(i,n)}) = z_{i+1}. \tag{27}$$

Podemos representar F_n con un circuito de tamaño constante, independiente de n. Como M es oblivious, las posiciones de las cabezas de entrada y trabajo de M solo dependen de n y del número de paso del cómputo de M con entrada x. Entonces todo el circuito C_n cumple $|C_n| = O(n + t(n))$ y $C_n(x) = M(x)$. La Figura 32 muestra el esquema del circuito C_n que construimos a partir de M. □

Teorema 28. $\mathbf{P} \not\supseteq \mathbf{P}_{/\mathbf{poly}}$.

Demostración. Recordemos del Ejemplo 24 que si $\mathcal{L} \subseteq \{1^n : n \in \mathbb{N}\}$ entonces $\mathcal{L} \in \mathbf{P}_{/\mathbf{poly}}$. Podemos tomar un $\mathcal{L} \subseteq \{1^n : n \in \mathbb{N}\}$ indecidible, por ejemplo

$$\mathcal{H} = \{1^n : x \text{ es la } n\text{-ésima cadena y } halt(x) = 1\}.$$

Acá numeramos las cadenas en orden de longitud y lexicográfico, es decir,

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots.$$

Entonces, tenemos que $\mathcal{H} \in \mathbf{P}_{/\mathbf{poly}} \setminus \mathbf{P}$. □

8.2. El Teorema de Karp-Lipton

Recordemos que podemos codificar fórmulas booleanas (con constantes 0 y 1, recordar la gramática (6)) en binario. Cada fórmula booleana de tamaño n tiene a lo sumo n variables y las suponemos numeradas $x_1, \ldots, x_i$, con $i \leq n$. Recordemos que usamos $\varphi(x_1, \ldots, x_n)$ para marcar que las variables de φ están entre $x_1, \ldots, x_n$. Para $v \in \{0,1\}^n$, notamos $\varphi(v)$ a la fórmula booleana que resulta de reemplazar x_i por la constante $v(i-1) \in \{0,1\}$, para $i = 1, \ldots, n$. Tenemos que $v \models \varphi(x_1, \ldots, x_n)$ sii $\models \varphi(v)$.

Consideramos a partir de ahora circuitos booleanos con n entradas y m salidas, que representan funciones $\{0,1\}^n \to \{0,1\}^m$. El siguiente resultado nos dice cómo encontrar valuaciones a partir de SAT.

Si $\varphi(x_1, \ldots, x_n)$ es una fórmula con variables entre $x_1, \ldots, x_n$ y $b \in \{0,1\}$, entonces representamos

$$\varphi(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

a la fórmula que resulta de reemplazar todas las apariciones de x_i por la constante b. Por ejemplo, para $\varphi(x_1, x_2, x_3) = (x_1 \wedge \neg x_2) \vee (x_3 \wedge x_2)$ tenemos que $\varphi(x_1, 1, x_3) = (x_1 \wedge \neg 1) \vee (x_3 \wedge 1)$ y $\varphi(x_1, 0, x_3) = (x_1 \wedge \neg 0) \vee (x_3 \wedge 0)$. Recordemos que identificamos valuaciones con cadenas en $\{0,1\}^*$.

Proposición 37. *Sea $(C_n)_{n\in\mathbb{N}}$ una familia de circuitos de tamaño $S(n)$ tal que*[14] *$C_n(\varphi) = \chi_{\mathsf{SAT}}(\varphi)$ para toda fórmula booleana $\varphi = \varphi(x_1, \ldots, x_n)$ con $|\varphi| = n$. Entonces existe una familia de circuitos $(C'_n)_{n\in\mathbb{N}}$ de tamaño polinomial en $S(n)$ tal que para todo n: C'_n tiene n salidas y para toda fórmula booleana $\varphi = \varphi(x_1, \ldots, x_n)$ con $|\varphi| = n$, tenemos $\varphi(C'_n(\varphi)) = \chi_{\mathsf{SAT}}(\varphi)$.*

Demostración. Si $\varphi(x_1, \ldots, x_n)$ es satisfacible, entonces

$$\begin{aligned} \varphi(1, x_2 \ldots, x_n) \in \mathsf{SAT} &\implies \exists v \in \{0,1\}^{n-1}\ 1v \models \varphi(x_1, \ldots, x_n) \\ \varphi(1, x_2 \ldots, x_n) \notin \mathsf{SAT} &\implies \exists v \in \{0,1\}^{n-1}\ 0v \models \varphi(x_1, \ldots, x_n) \end{aligned}$$

Si $\varphi(x_1, \ldots, x_n)$ es satisfacible, entonces

$$\varphi(\overbrace{\chi_{\mathsf{SAT}}(\varphi(1, x_2, \ldots, x_n))}^{\in\{0,1\}}, x_2, \ldots, x_n)$$

es satisfacible. Podemos seguir adelante con el mismo razonamiento. Si $\varphi(x_1, \ldots, x_n)$ es satisfacible, entonces

$$\varphi(\overbrace{\chi_{\mathsf{SAT}}(\varphi(1, x_2, \ldots, x_n))}^{b\in\{0,1\}}, \overbrace{\chi_{\mathsf{SAT}}(\varphi(b, 1, x_3 \ldots, x_n))}^{\in\{0,1\}}, x_3, \ldots, x_n)$$

es satisfacible. Y así sucesivamente. Este procedimiento encuentra una valuación para φ, en caso de que sea satisfacible (en caso de que no, construye una valuación que, obviamente, no satisfará a φ).

Dada $\varphi = \varphi(x_1, \ldots, x_n)$ de tamaño n y el circuito C_n tal que $C_n(\varphi) = \chi_{\mathsf{SAT}}(\varphi)$, definimos el circuito C'_n con entrada φ y salidas $r_1, \ldots, r_n$ ($r_i \in \{0,1\}$) como en el esquema de la Figura 33. Es importante notar que los circuitos que reemplazan apariciones de variables de φ por constantes 0 o 1 (que a su vez pueden venir de la salida de un subcircuito) tienen tamaño polinomial. Así, tenemos que $\varphi \in \mathsf{SAT}$ sii $r_1 \ldots r_n \models \varphi$ sii $\models \varphi(C'_n(\varphi))$. □

[14] Notamos $C_n(\varphi)$ en vez de $C_n(\langle\varphi\rangle)$ y lo mismo para C'_n.

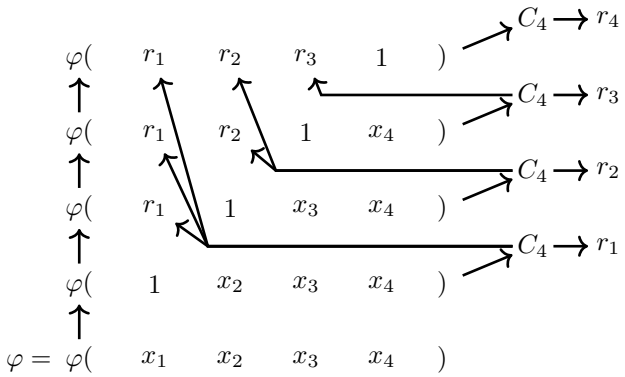


Figura 33: Un esquema para el circuito que encuentra una valuación para φ, con variables x_1, x_2, x_3, x_4, cuando φ es satisfacible. La entrada es una codificación de φ y la salida, que representa la valuación para φ, es $r_1, \ldots, r_4$. El circuito C_4 es tal que $C_4(\varphi) = \chi_{\mathsf{SAT}}(\varphi)$. Este esquema no muestra los circuitos de tamaño polinomial que hacen cada transformación (por ejemplo el que reemplaza la variable x_1 de φ por 1) sino solamente el flujo de la información.

Teorema 29 (Karp-Lipton)**.** *Si* $\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$*, entonces* $\mathbf{PH} = \mathbf{\Sigma_2^p}$.

Demostración. Para la fórmula booleana posiblemente con constantes (gramática (6))

$$\varphi = \varphi(x_1, \ldots, x_i, y_1, \ldots, y_j)$$

con variables $x_1, \ldots, x_i, y_1, \ldots, y_j$, y $\bar{u} = u_1, \ldots, u_i \in \{0,1\}^*$ notamos

$$\varphi_{\bar{u}} = \varphi_{\bar{u}}(y_1, \ldots, y_j) = \varphi(u_1, \ldots, u_i, y_1, \ldots, y_j).$$

Por la Proposición 33 sabemos que $\Pi_2\mathsf{SAT}$ es $\mathbf{\Pi_2^p}$**-completo**. Por el Ejercicio 15 alcanza con probar $\mathbf{\Pi_2^p} = \mathbf{\Sigma_2^p}$, o, equivalentemente, con probar que $\Pi_2\mathsf{SAT} \in \mathbf{\Sigma_2^p}$. Supongamos que $\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$. Existe un polinomio p y una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ de tamaño $p(n)$ tal que para toda fórmula booleana φ de tamaño n

$$\varphi \in \mathsf{SAT} \quad \text{sii} \quad \exists \bar{v} \in \{0,1\}^n \ \bar{v} \models \varphi \quad \text{sii} \quad \exists \bar{v} \in \{0,1\}^n \ \models \varphi(\bar{v}) \quad \text{sii} \quad C_n(\varphi) = 1.$$

Supongamos $\varphi = \varphi(x_1, \ldots, x_i, y_1, \ldots, y_j)$ una fórmula booleana de tamaño n y llamemos $\bar{x} = x_1, \ldots, x_i$ y $\bar{y} = y_1, \ldots, y_j$. Tenemos que

$$\forall \bar{u} \in \{0,1\}^i \left(\underbrace{\exists \bar{v} \in \{0,1\}^j \ \models \varphi_{\bar{u}}(\bar{v})}_{\varphi_{\bar{u}} = \varphi_{\bar{u}}(\bar{y}) \in \mathsf{SAT}} \iff C_n(\varphi_{\bar{u}}(\bar{y})) = 1 \right).$$

Luego

$$\begin{aligned}
\forall \bar{x} \exists \bar{y} \ \varphi(\bar{x}, \bar{y}) \in \Pi_2\mathsf{SAT} \quad & \text{sii} \quad \forall \bar{u} \in \{0,1\}^i \overbrace{\exists \bar{v} \in \{0,1\}^j \ \models \varphi(\bar{u}, \bar{v})}^{\varphi_{\bar{u}} \in \mathsf{SAT}} \\
& \text{sii} \quad \forall \bar{u} \in \{0,1\}^i \ C_n(\varphi_{\bar{u}}) = 1 \\
& \text{sii} \quad \forall \bar{u} \in \{0,1\}^i \ \models \varphi_{\bar{u}}(C'_n(\varphi_{\bar{u}})) \qquad (28) \\
& \text{sii} \quad \underbrace{\exists c \in \{0,1\}^{q(n)} \forall \bar{u} \in \{0,1\}^i \ \models \varphi_{\bar{u}}(R_c(\varphi_{\bar{u}}))}_{\exists c \in \{0,1\}^{q(n)} \forall \bar{u} \in \{0,1\}^i \ [\text{Polinomial}] \ \in \ \mathbf{\Sigma_2^p}}, \qquad (29)
\end{aligned}$$

donde $|\varphi(\bar{x}, \bar{y})| = |\varphi_{\bar{u}}(\bar{y})| = n$; en (28), C'_n es un circuito de tamaño polinomial en n cuya existencia surge de la Proposición 37 y tal que $\psi(C'_n(\psi)) = \chi_{\mathsf{SAT}}(\psi)$ para toda ψ de tamaño n; R_c es el circuito

representado por $c \in \{0,1\}^*$; y q es un polinomio tal que $|\langle C'_n \rangle| \leq q(n)$ Observemos que podemos decidir $\varphi_{\bar{u}}(R_c(\varphi_{\bar{u}}))$ en tiempo polinomial y por lo tanto la fórmula de (29) tiene la forma adecuada para la clase $\mathbf{\Sigma_2^P}$. □

Como se cree que **PH** no colapsa en ningún nivel, el Teorema 29 nos da un indicio de que $\mathbf{NP} \not\subseteq \mathbf{P_{/poly}}$.

8.3. Tamaño y profundidad de circuitos

Definición 30. Sea $f : \{0,1\}^n \to \{0,1\}$. Definimos

$$\text{minSize}(f) = \text{mín}\{|C| \colon C \text{ tiene } n \text{ entradas y } \forall \bar{x}\ C(\bar{x}) = f(\bar{x})\}$$

El siguiente resultado nos da una cota superior para minSize.

Proposición 38. $\text{minSize}(f) = O(n2^n)$ *para toda* $f : \{0,1\}^n \to \{0,1\}$.

Demostración. Ya lo probamos en la Proposición 8 para CNF. Otra prueba: podríamos describir a f en DNF (forma normal disyuntiva) con

$$f(x_1, \ldots, x_n) = \bigvee_{\substack{(y_1,\ldots,y_n) \in \{0,1\}^n \colon \\ f(y_1,\ldots,y_n)=1}} \bigwedge_{1 \leq i \leq n} x_i = y_i,$$

donde la subfórmula $x_i = y_i$ representa x_i si $y_i = 1$ y $\neg x_i$ en caso contrario. Podemos ver que la fórmula se representa con un circuito de tamaño $O(n2^n)$. □

El siguiente resultado nos da una cota inferior para minSize.

Teorema 30. *Para todo* $n > 1$, *existe* $f : \{0,1\}^n \to \{0,1\}$ *tal que* $\text{minSize}(f) > 2^n/4n$.

Demostración. Supongamos un circuito C de tamaño a lo sumo S. De (26) sabemos que podemos codificar C con a lo sumo $4 \cdot S \cdot \log S$ bits.

Para cualquier función $S : \mathbb{N} \to \mathbb{N}$, definimos $K_{n,S}$ como el conjunto de circuitos con n entradas de tamaño a lo sumo $S(n)$. Claramente tenemos que $\#K_{n,S} \leq 2^{4S(n)\log S(n)}$. Tomemos $S(n) = 2^n/4n$. Entonces codificamos cada $C \in K_{n,S}$ con a lo sumo:

$$4 \cdot S(n) \cdot \log S(n) = 4\frac{2^n}{4n} \log \frac{2^n}{4n} < \frac{2^n}{n} \log 2^n = 2^n$$

bits y por lo tanto $\#K_{n,S} < 2^{2^n}$. Como la cantidad de funciones $\{0,1\}^n \to \{0,1\}$ es 2^{2^n}, existe una función $f : \{0,1\}^n \to \{0,1\}$ tal que $\text{minSize}(f) > 2^n/4n$. □

Definición 31. La **profundidad** de un circuito C, notado $\text{prof}(C)$, es la longitud del camino más largo desde alguna entrada hasta alguna salida.

Ejemplo 28. La Figura 34 muestra la profundidad de dos circuitos.

Definición 32. Una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ tiene profundidad $d(n)$ si $\text{prof}(C_n) \leq d(n)$ para todo n.

Ejemplo 29. El lenguaje $\{1^n \colon n \geq 1\}$ es decidible por una familia de circuitos de profundidad $O(\log n)$, como muestra la Figura 34.

Vamos a considerar circuitos con $\wedge$ y $\vee$ con *fan-in* arbitrario. Esto permite construir circuitos con profundidades más bajas.

Ejemplo 30. Existe una familia de circuitos con $\wedge$ de *fan-in* arbitrario de profundidad $O(1)$ que decide $\{1^n \colon n \geq 1\}$, como muestra la Figura 35.

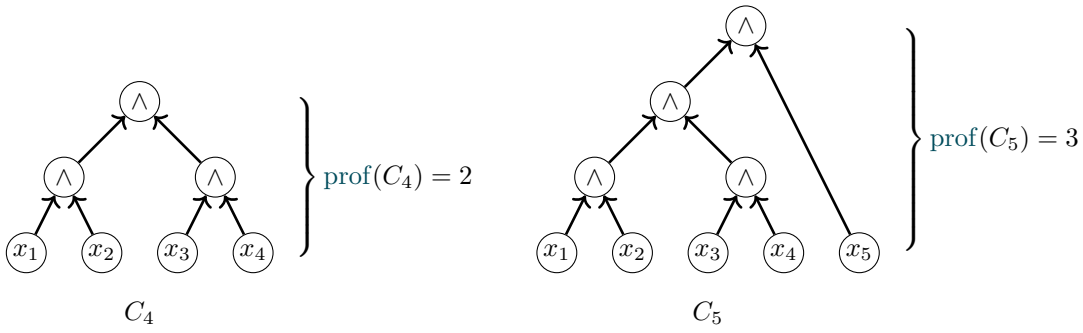


Figura 34: Circuitos de profundidad logarítmica.

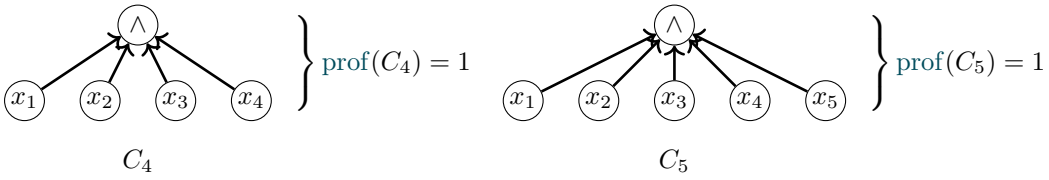


Figura 35: $\{1^n : n \geq 1\}$ es decidible por una familia de circuitos de profundidad constante, si permitimos $\wedge$ de *fan-in* arbitrario.

8.4. Las clases $\mathbf{NC}_{\text{nu}}$ y $\mathbf{AC}_{\text{nu}}$

Clase de complejidad: $\mathbf{NC}^d_{\text{nu}}$ y $\mathbf{NC}_{\text{nu}}$

- $\mathbf{NC}^d_{\text{nu}}$ es la clase de lenguajes decidible por una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ tal que $|C_n|$ es polinomial en n y $\text{prof}(C_n) = O(\log^d n)$
- $\mathbf{NC}_{\text{nu}} = \bigcup_{i\geq 0} \mathbf{NC}^i_{\text{nu}}$

Clase de complejidad: $\mathbf{AC}^d_{\text{nu}}$ $\mathbf{AC}_{\text{nu}}$

- $\mathbf{AC}^d_{\text{nu}}$ se define igual que $\mathbf{NC}^i_{\text{nu}}$ pero permitimos nodos $\wedge$ y $\vee$ con *fan-in* arbitrario.
- $\mathbf{AC}_{\text{nu}} = \bigcup_{i\geq 0} \mathbf{AC}^i_{\text{nu}}$

El subíndice 'nu' es por 'no uniforme', e indica que no hay requerimiento de uniformidad. En §8.5 veremos el caso uniforme.

Notemos que la clase $\mathbf{NC}^0_{\text{nu}}$ es muy limitada porque la salida de un circuito en $\mathbf{NC}^0_{\text{nu}}$ solo depende de una cantidad constante de bits de la entrada. En cambio, $\mathbf{AC}^0_{\text{nu}}$ no tiene esa misma limitación. Como vimos en el Ejemplo 29, $\{1^n : n \geq 1\} \in \mathbf{AC}^0_{\text{nu}}$.

Problema: Cantidad impar de 1s

$$\text{PARITY} = \{x : x \text{ tiene una cantidad impar de 1s}\}$$

Proposición 39. $\text{PARITY} \in \mathbf{NC}^1_{\text{nu}}$.

Demostración. Sumamos (módulo 2) todos los bits de x, como muestra la Figura 36. □

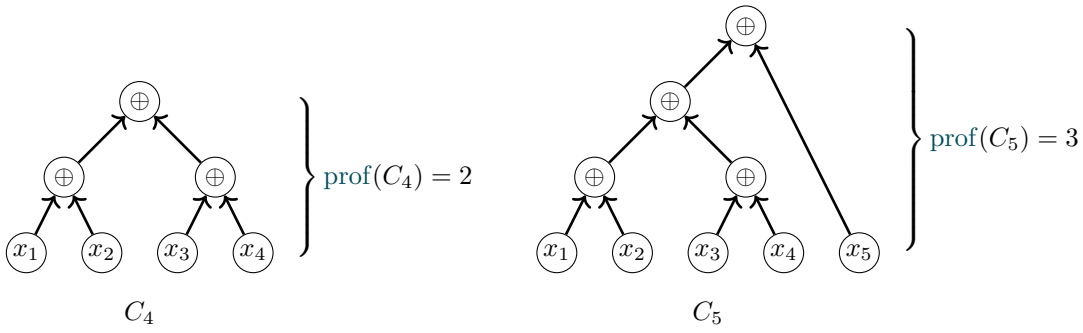


Figura 36: Circuitos para sumar en binario los bits de la entrada. $\oplus$ representa el 'o exclusivo', ver Figura 28-*a*. En general, $\text{prof}(C_n) = O(\log n)$.

Entrada: $x_n, \dots, x_1, y_n, \dots, y_1, z_{n+1}, z_n, \dots, z_1$.

carry $\to$	$\dots$	1	1	1			1	
$x =$	$\dots$		1	0	1	1	0	1
$y =$	$\dots$		0	1	1	0	0	1
$z =$	$\dots$	1	0	1	0	1	1	0
		$\downarrow$			$\downarrow$			
		$i+1$			j			

Figura 37: En cada paso el *carry* en la suma en binario solo depende de la entrada.

La demostración de la siguiente proposición cae fuera del alcance de este curso.

Proposición 40. PARITY $\notin \mathbf{AC}^0_{\text{nu}}$.

En cambio, sí tenemos herramientas para probar lo siguiente:

Proposición 41. SUMA $\in \mathbf{AC}^0_{\text{nu}}$.

Demostración. Recordemos el Ejemplo 25. Calculamos el *carry* c_i de esta forma (ver Figura 37):

$$c_1 = 0 \quad \text{y} \quad c_{i+1} = \bigvee_{i \geq j \geq 1} \left((x_j \wedge y_j) \wedge \bigwedge_{i \geq k > j} (x_k \vee y_k) \right)$$

Notemos que, gracias a que podemos usar la conjunción y disyunción arbitraria, c_{i+1} solo depende de la entrada y no de c_i. En cambio, en (25) no podíamos hacer lo mismo porque solo teníamos conjunciones y disyunciones binarias. Cada circuito para calcular c_i tiene profundidad 3. Con nodos internos s_i del circuito calculamos la suma como lo hicimos en (24): $s_i = x_i \oplus y_i \oplus c_i$ y $s_{n+1} = c_{n+1}$. Todo esto lo hacemos con un circuito de profundidad constante. Finalmente, la salida es

$$\bigwedge_{n+1 \geq j \geq 1} z_i = s_i$$

y también lo calculamos con un circuito de profundidad constante. □

Proposición 42. $\mathbf{NC}^d_{\text{nu}} \subseteq \mathbf{AC}^d_{\text{nu}} \subseteq \mathbf{NC}^{d+1}_{\text{nu}}$. *Por lo tanto,* $\mathbf{AC}_{\text{nu}} = \mathbf{NC}_{\text{nu}}$.

Demostración. $\mathbf{NC}^d_{\text{nu}} \subseteq \mathbf{AC}^d_{\text{nu}}$ es trivial. Veamos $\mathbf{AC}^d_{\text{nu}} \subseteq \mathbf{NC}^{d+1}_{\text{nu}}$. Supongamos que $\mathcal{L} \in \mathbf{AC}^d_{\text{nu}}$. Existe una familia de circuitos con $\wedge, \vee$ de *fan-in* arbitrario $(C_i)_{i \in \mathbb{N}}$ que decide $\mathcal{L}$, $|C_n| = O(n^c)$ $\text{prof}(C_n) = O(\log^d n)$. Simulamos el *fan-in* arbitrario con un circuito con *fan-in* binario de profundidad $O(\log n)$. Definimos C'_n como el reemplazo en C_n de todo nodo de *fan-in* arbitrario con el

subcircuito de profundidad $O(\log n^c) = O(\log n)$. Es claro que $(C'_i)_{i\in\mathbb{N}}$ decide $\mathcal{L}$ y que tiene tamaño polinomial en n. Como cualquier camino desde la entrada a la salida en C_n tiene profundidad $O(\log^d)$, resulta que en C'_n tiene profundidad $O(\log n \cdot \log^d n) = O(\log^{d+1} n)$ y esto prueba que $\mathcal{L} \in \mathbf{NC}^{d+1}_{\text{nu}}$. □

8.5. Uniformidad: las clases NC y AC

Recordemos la demostración del Teorema 28. Puede parecer extraño que lenguajes indecidibles estén en $\mathbf{P}_{/\mathbf{poly}}$. El origen de este fenómeno es la no uniformidad del modelo de circuitos. Es decir, la Definición 28 de familia de circuitos es solo existencial: pide que exista la familia de circuitos, pero no impone ninguna condición de cómo deben se generada la familia.

Definición 33. Una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ es **P-uniforme** si existe una máquina determinística tal que M corre en tiempo polinomial y $M(1^n) = \langle C_n\rangle$.

Entonces, una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ es **P**-uniforme cuando puede ser generada eficientemente, es decir, existe una máquina determinística que corre en tiempo polinomial y que con entrada 1^n devuelve una codificación de C_n.

Teorema 31. *$\mathcal{L}$ es decidible por una familia de circuitos **P**-uniforme sii $\mathcal{L} \in \mathbf{P}$.*

Demostración. Veamos $\subseteq$. Sea M una máquina determinística que corre en tiempo polinomial tal que para todo n, tenemos que $M(1^n) = \langle C_n\rangle$ y para todo $x \in \{0,1\}^n$: $x \in \mathcal{L}$ sii $C_n(x) = 1$. Definimos la máquina determinística M' que, con entrada x, hace esto:

Simular $M(1^{|x|}) = \langle C\rangle$
Evaluar $C(x)$ y devolver su salida

La simulación y la evaluación $C(x)$ lleva tiempo polinomial. Como $M'(x) = C_{|x|}(x) = \chi_{\mathcal{L}}(x)$, concluimos que $\mathcal{L} \in \mathbf{P}$.

Para $\supseteq$, seguimos la demostración del Teorema 27, que prueba que $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$. La familia de circuitos que se construye es, de hecho, **P**-uniforme. □

Recordemos la definición de una función computable implícitamente en **L** de la Definición 23.

Definición 34. Una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ es **L-uniforme** si existe una función $f : \{0,1\}^* \to \{0,1\}^*$ computable implícitamente en **L** tal que $f(1^n) = \langle C_n\rangle$.

Ejercicio 17. Probar que $\mathcal{L}$ es decidible por una familia de circuitos **L**-uniformes sii $\mathcal{L} \in \mathbf{P}$. La inclusión $\subseteq$ es consecuencia del hecho de que $\mathbf{L} \subseteq \mathbf{P}$ y del Teorema 31. Para $\supseteq$, repasemos la demostración de $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$ del Teorema 27. Observemos que, si bien el tamaño del circuito C_n de la Figura 32 es polinomial,[15] podemos computar su codificación a partir de 1^n implícitamente en **L**.

Clase de complejidad: NC

- $\mathbf{NC}^d$ es la clase de lenguajes decidibles por una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ **L**-uniforme tal que $|C_n|$ es polinomial en n y $\text{prof}(C_n) = O(\log^d n)$
- $\mathbf{NC} = \bigcup_{i\geq 0} \mathbf{NC}^i$

[15] La demostración del Teorema 27 considera una máquina oblivious, gracias a la Proposición 5, que tiene como hipótesis que el tiempo de cómputo sea una función construible en tiempo. Dado un polinomio t, existe una constante c suficientemente grande tal que $t'(n) = c + n^c > t(n)$. Entonces, t' es constuible en tiempo y también computable implícitamente en **L**.

Clase de complejidad: AC

- $\mathbf{AC}^d$ se define igual que $\mathbf{NC}^d$ pero permitimos nodos $\wedge$ y $\vee$ con *fan-in* arbitrario.
- $\mathbf{AC} = \bigcup_{i \geq 0} \mathbf{AC}^i$

Varios resultados que ya vimos adaptan al escenario de uniformidad. Por ejemplo, la Proposición 39 se convierte en

Proposición 43. PARITY $\in \mathbf{NC}^1$.

La Proposición 41 se transforma en

Proposición 44. SUMA $\in \mathbf{AC}^0$.

La Proposición 42 se convierte en

Proposición 45. $\mathbf{NC}^d \subseteq \mathbf{AC}^d \subseteq \mathbf{NC}^{d+1}$. *Por lo tanto,* $\mathbf{AC} = \mathbf{NC}$.

Proposición 46. $\mathbf{NC} \subseteq \mathbf{P}$.

Demostración. Es una consecuencia inmediata del Ejercicio 17 y de la definición de $\mathbf{NC}$. □

Un problema tiene una **solución paralela eficiente** si puede ser resuelto para entradas de tamaño n usando una computadora paralela con una cantidad polinomial ($n^{O(1)}$) de procesadores en tiempo polilogarítmico ($\log^{O(1)} n$).

Teorema 32. *$\mathcal{L}$ tiene una solución paralela eficiente sii $\mathcal{L} \in \mathbf{NC}$.*

Algunos ejemplos de problemas en $\mathbf{NC}$ son suma, multiplicación y división enteras,suma y multiplicación de matrices, determinante, inversa, rango, camino mínimo, y spanning tree mínimo en grafos, entre otros.

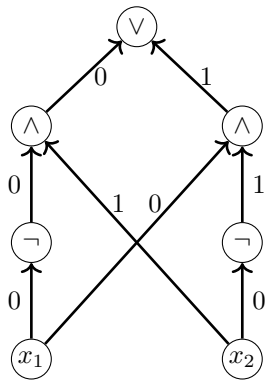
Teorema 33. $\mathbf{NC}^1 \subseteq \mathbf{L}$.

Demostración. Sea $\mathcal{L} \in \mathbf{NC}^1$ y sea $(C_n)_{n \in \mathbb{N}}$ una familia de circuitos tal que $|C_n|$ es polinomial en n, $\text{prof}(C_n) = O(\log n)$ y $x \in \mathcal{L}$ sii $C_{|x|}(x) = 1$, y sea M una máquina determinística que computa implícitamente en $\mathbf{L}$ la función $1^n \mapsto \langle C_n \rangle$ Definimos la función $g(1^n, x, u)$ recursivamente como sigue:

- si u es la codificación del k-ésimo ($1 \leq k \leq n$) nodo de entrada de C_n, devolver $x(k-1)$
- si u es la codificación de un nodo etiquetado con $*$ (con $* \in \{\wedge, \vee\}$) de C_n, con hijos v_1, v_2, devolver $g(1^n, x, v_1) * g(1^n, x, v_2)$
- si u es la codificación de un nodo etiquetado con $\neg$ de C_n, con hijo v, devolver $\neg g(1^n, x, v)$

Tenemos que $C_{|x|}$ sii $g(1^{|x|}, x, u_0) = 1$, donde u_0 es la codificación de la salida (*sink*) de $C_{|x|}$. Sea $n = |x|$. Podemos computar $g(1^n, x, u_0)$ en espacio $O(\log n)$ de esta manera. Para resolver la recursión recorremos el circuito C_n usando DFS. Cada camino de longitud i desde el nodo de salida se puede codificar con una palabra $\{0,1\}^i$, donde 0 significa 'hijo izquierdo' o 'único hijo', y 1 significa 'hijo derecho'. Por hipótesis, la profundidad del circuito es logarítmica, de modo que $i \in O(\log n)$. Entonces nos alcanza con espacio logarítmico para llevar cuenta del camino que estamos analizando. Para saber qué tipo de nodo es el que estamos analizando en cada momento, y por lo tanto también para saber la cantidad de hijos que tiene, simulamos $M(1^n)$. Por hipótesis, esto también usa espacio $O(\log n)$. Ver el Ejemplo 31. □

Ejemplo 31. Veamos la idea de cómo evaluar un circuito usando eficientemente el espacio. Analicemos el circuito de la Apartado 8.5. Hacemos un recorrido DFS por el circuito, empezando desde el nodo de salida. Entre [ ] hay 1 bit que depende de los valores de x_1 y x_2. Ahí vamos calculando las operaciones intermedias (conjunciones, negaciones, disyunciones). Cada estado usa espacio $O(h)$, donde h es la profundidad del circuito.



ε	$0[\neg x_1]1$	
0	$0[\neg x_1]1[x_2]$	$[\neg x_1 \wedge x_2]1[x_1]$
00	$01[\neg x_1][x_2]$	$[\neg x_1 \wedge x_2]1[x_1]1$
000	$0[\neg x_1 \wedge x_2]$	$[\neg x_1 \wedge x_2]1[x_1]10$
$000[x_1]$	$[\neg x_1 \wedge x_2]$	$[\neg x_1 \wedge x_2]1[x_1]10[x_2]$
$00[x_1]$	$[\neg x_1 \wedge x_2]1$	$[\neg x_1 \wedge x_2]1[x_1]1[\neg x_2]$
$0[\neg x_1]$	$[\neg x_1 \wedge x_2]10$	$[\neg x_1 \wedge x_2][x_1 \wedge \neg x_2]$
	$[\neg x_1 \wedge x_2]10[x_1]$	$[(\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)]$

Figura 38: Izquierda: un circuito con los ejes etiquetados. Derecha: la evaluación que usa espacio del orden de la profundidad del circuito. Notar que no entra todo el circuito en memoria porque tienen tamaño polinomial; hay que ir pidiendo a M los nodos a medida que se necesitan para el recorrido de modo de usar solamente espacio logarítmico

Teorema 34. *Sea $\mathcal{L} \in \mathbf{NSpace}(S(n))$. Existe una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ con compuertas $\wedge$ y $\vee$ de* fan-in *arbitrario y una máquina determinística M tal que para todo $n \geq 1$ y $x \in \{0,1\}^n$ tenemos que $x \in \mathcal{L}$ sii $C_n(x) = 1$, $M(1^n) = \langle C_n \rangle$, $|C_n| = 2^{O(S(n))}$ y $\text{prof}(C_n) = O(S(n))$.*

Demostración. Sea N una máquina no-determinística que usa espacio $O(S(n))$ tal que $x \in \mathcal{L}$ sii existe un cómputo aceptador de N con entrada x. Consideramos el grafo de configuraciones $G_{N,x}$ de la Definición 21, que tiene $m = 2^{c \cdot S(|x|)}$ nodos, para alguna constante c. Sea $A_x \in \{0,1\}^{m\times m}$ la matriz de adyacencia de $G_{N,x}$. Cada fila y columna de A_x representa una configuración del cómputo de N con entrada x de modo tal que$(A_x)_{ij} = 1$ sii i evoluciona en un paso en j. Definimos $B_x = A_x \vee I$, donde I es la matriz identidad de dimensión $m \times m$ y el $\vee$ se define coordenada a coordenada. Tenemos que $(B_x)_{ij} = 1$ sii j es alcanzable desde i en a lo sumo 1 paso. Consideramos una multiplicación $\otimes$ de matrices donde $\vee$ juega de $+$ y $\wedge$ juega de $\cdot$, es decir, para $C, D \in \{0,1\}^{m\times m}$:

$$(C \otimes D)_{ij} = \bigvee_{1\leq k\leq m} C_{ik} \wedge D_{kj}. \tag{30}$$

Entonces, $(B_x^\ell)_{ij} = 1$ sii j es alcanzable desde i en a lo sumo ℓ pasos.

Representamos matrices de dimensión $m \times m$ con circuitos: cada nodo es una posición de la matriz. Pensemos en una matriz de $m \times m$ como una tira de m^2 nodos. Sea $n = |x|$. Construimos C_n con entrada x de esta forma: Primero calculamos A_x; esto lo logramos con un circuito de profundidad constante (recordemos que disponemos los nodos de la matriz en un mismo nivel del circuito). Luego, en el siguiente nivel, calculamos B_x, que también lo hacemos con un circuito de profundidad constante. A continuación (es decir, más 'arriba' en el circuito), calculamos $D_x = B_x^m = B_x^{2^{c\cdot S(n)}}$ (recordar que $m = 2^{c\cdot S(n)}$ es la cantidad de nodos de $G_{N,x}$. Esto nos toma profundidad logarítmica en la dimensión $m = 2^{c\cdot S(n)}$, es decir profundidad $O(S(n))$ (usamos la multiplicación en forma de árboles; ejemplo: $B_x^{2^3} = ((B_x \cdot B_x) \cdot (B_x \cdot B_x)) \cdot ((B_x \cdot B_x) \cdot (B_x \cdot B_x)))$. Si suponemos que la configuración inicial es 1 y la final es la 2, entonces devolvemos $(D_x)_{1,2}$. $(D_x)_{1,2} = 1$ sii hay un cómputo aceptador de N a partir de x, es decir sii $N(x) = 1$. Luego, $C_n(x) = 1$ sii N acepta x, $\text{prof}(C_n) = O(S(n))$ y $|C_n| = 2^{O(S(n))}$. Las matrices A_x, B_x y D_x dependen de x, pero el circuito para calcularlas es el mismo para todo $x \in \{0,1\}^n$. Entonces, la definición del circuito solo depende de $n = |x|$. La construcción es uniforme, es decir, existe una máquina determinística M tal que $M(1^n) = \langle C_n \rangle$. Ver Figura 39. □

Corolario 8. $\mathbf{NL} \subseteq \mathbf{AC}^1$.

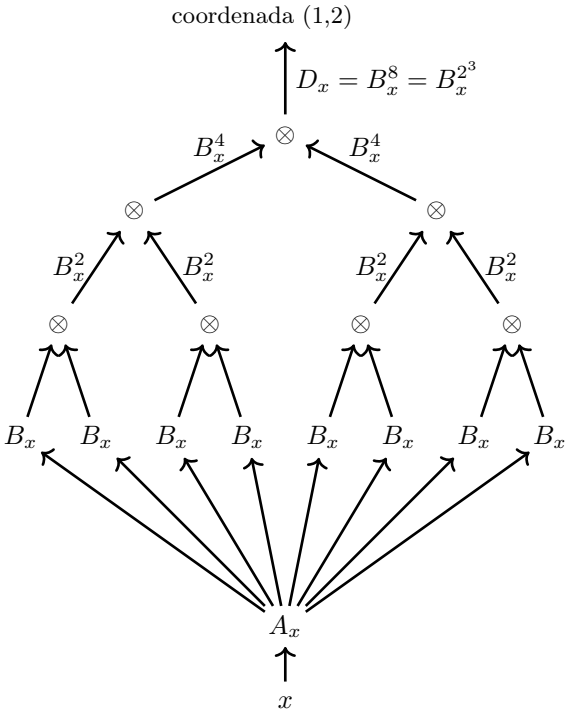


Figura 39: El circuito C_n de la demostración del Teorema 34. Los nodos $\times$ representan multiplicación de matrices y son de profundidad constante, gracias al *fan-in* arbitrario (ver (30)). El nodo A_x representa una fila de nodos que representa la matriz $A_x \in \{0,1\}^{2^3 \times 2^3}$; lo mismo para cada nodo B_x y para D_x. En general, $|A_x| = |B_x| = |D_x| = (2^{c \cdot S(n)})^2$, el tamaño del circuito es $2^{O(S(n))}$ y la profundidad es $O(S(n))$.

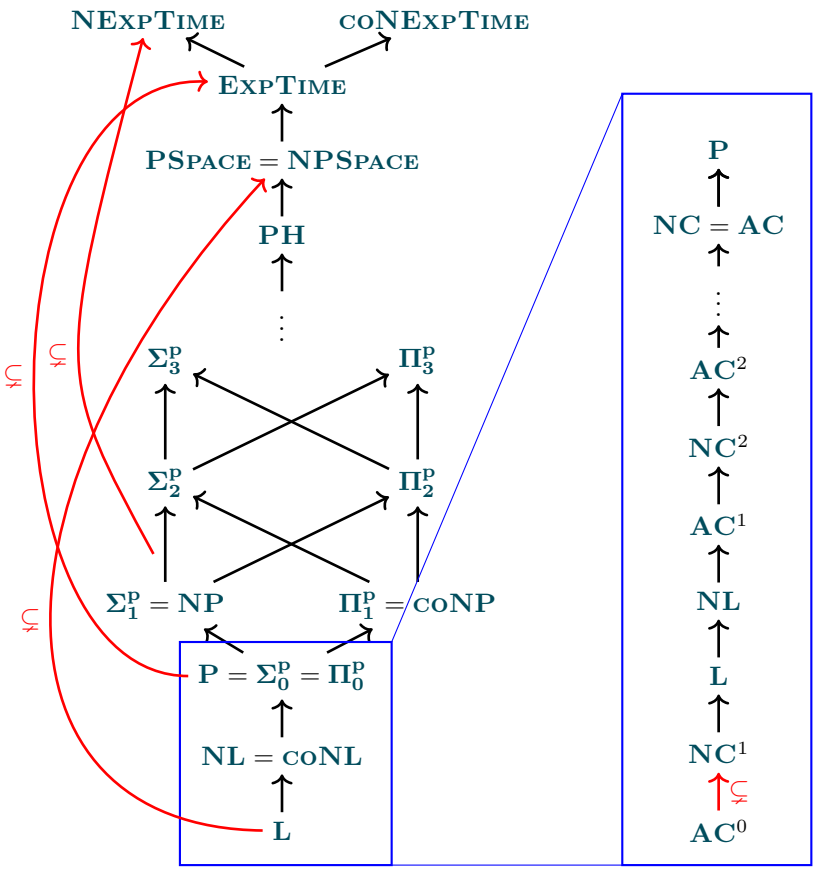


Figura 40: Las clases de complejidad que vimos en el curso. De la jerarquía de la derecha, se conoce la inclusión estricta $\mathbf{AC}^0 \subsetneq \mathbf{NC}^1$ (en rojo, consecuencia de las Proposiciones 40 y 43), pero se conjetura que toda esa jerarquía es estricta. En la parte derecha aparecen algunas otras inclusiones estrictas en rojo que ya habíamos visto en capítulos anteriores y que son consecuencia de los resultados de las jerarquías de tiempos y espacios.

Demostración. Sea $\mathcal{L} \in \mathbf{NSpace}(\log n)$. Del Teorema 34 sabemos que existe una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ con compuertas $\wedge$ y $\vee$ de *fan-in* arbitrario y una máquina determinística M tal que para todo $n \geq 1$ y $x \in \{0,1\}^n$ tenemos que $x \in \mathcal{L}$ sii $C_n(x) = 1$, $M(1^n) = \langle C_n \rangle$, $|C_n| = 2^{O(\log n)} = n^{O(1)}$ y $\text{prof}(C_n) = O(\log n)$. Se puede ver que, de hecho, $1^n \mapsto \langle C_n \rangle$ es computable implícitamente en $\mathbf{L}$ por M. □

Las clases de complejidad que vimos se resumen en la Figura 40.

8.6. P-completitud

La pregunta $\mathbf{NC} \stackrel{?}{=} \mathbf{P}$ está abierta y, gracias al Teorema 32, se puede traducir en la pregunta "¿cualquier problema factible tiene una solución paralela eficiente?". Sabemos que $\mathbf{NC}^1 \subsetneq \mathbf{PSpace}$ porque $\mathbf{L} \subsetneq \mathbf{PSpace}$. Entonces, $\mathbf{NC}^1 \neq \mathbf{PSpace}$, pero no sabemos $\mathbf{NC} \stackrel{?}{=} \mathbf{P}$; ni siquiera sabemos $\mathbf{NC}^1 \stackrel{?}{=} \mathbf{PH}$.

Clase de complejidad: P-completo

$\mathcal{L}$ es **P-completo** si $\mathcal{L} \in \mathbf{P}$ y $\mathcal{L}' \leq_\ell \mathcal{L}$ para todo $\mathcal{L}' \in \mathbf{P}$.

Si apostamos a que $\mathbf{NC} \neq \mathbf{P}$, entonces los problemas **P-completos** son los que, a pesar de ser factibles, no tienen una solución paralela eficiente.

Problema: Evaluación de un circuito

$$\text{CIRC-EVAL} = \{\langle C, x\rangle : \begin{array}{l} C \text{ es un circuito con } n \text{ entradas con única salida,} \\ x \in \{0,1\}^n, \text{ y } C(x) = 1 \end{array} \}$$

Proposición 47. CIRC-EVAL *es* **P-completo**.

Demostración. Es claro que CIRC-EVAL $\in \mathbf{P}$. Supongamos $\mathcal{L} \in \mathbf{P}$ y sea M una máquina determinística que corre en tiempo polinomial y $\mathcal{L}(M) = \mathcal{L}$. Por el Ejercicio 17 sabemos que $\mathcal{L}$ es decidible por una familia de circuitos **L**-uniforme. Entonces existe una función $f : \{0,1\}^* \to \{0,1\}^*$ computable implícitamente en **L** tal que $f(1^n) = \langle C_n\rangle$ y satisface que $x \in \mathcal{L}$ sii $C_{|x|}(x) = 1$ sii $\langle f(1^{|x|}), x\rangle \in$ CIRC-EVAL. Como $x \mapsto \langle f(1^{|x|}), x\rangle$ es computable implícitamente en **L** concluimos que $\mathcal{L} \leq_\ell$ CIRC-EVAL. □

Ejercicio 18. Probar que si $\mathcal{L} \in \mathbf{NC}$ y $\mathcal{L}' \leq_\ell \mathcal{L}$, entonces $\mathcal{L}' \in \mathbf{NC}$.

Teorema 35. *Supongamos que* $\mathcal{L}$ *es* **P-completo**.

1. $\mathcal{L} \in \mathbf{NC}$ *sii* $\mathbf{P} = \mathbf{NC}$
2. $\mathcal{L} \in \mathbf{L}$ *sii* $\mathbf{P} = \mathbf{L}$

Demostración. El ítem 2 queda como ejercicio. La implicación $\Leftarrow$ del ítem 1 es trivial. Veamos la implicación $\Rightarrow$ del ítem 1,

Como $\mathcal{L} \in \mathbf{NC}$, existe e y una familia de circuitos $(C_n)_{n\in\mathbb{N}}$ **L**-uniforme tal que $|C_n| = O(n^e)$, $\text{prof}(C_n) = O(\log^e n)$, y para todo $x \in \{0,1\}^n$ tenemos que $C_n(x) = 1$ sii $x \in \mathcal{L}$. Sea $\mathcal{L}' \in \mathbf{P}$. Como $\mathcal{L}$ es **P-completo**, existe $f' : \{0,1\}^* \to \{0,1\}^*$ computable implícitamente en **L** tal que para todo $x \in \{0,1\}^n$ tenemos que $x \in \mathcal{L}'$ sii $f'(x) \in \mathcal{L}$. Por la Definición 23, existe c tal que $|f'(x)| \leq c \cdot n^c$. Por simplicidad, supongamos que $|f'(x)| = c \cdot n^c$. Como $\mathbf{L} \subseteq \mathbf{NC}$, existe d y una familia de circuitos $(C'_n)_{n\in\mathbb{N}}$ **L**-uniforme tal que $|C'_n| = O(n^d)$, $\text{prof}(C'_n) = O(\log^d n)$, y para todo $x \in \{0,1\}^n$ $C'_n(x) = f'(x)$ En realidad $\mathbf{L} \subseteq \mathbf{NC}$ lo vimos para lenguajes pero vale para cualquier función computable implícitamente en **L**, como f' [16]. Entonces

$$x \in \mathcal{L}' \quad \text{sii} \quad f'(x) \in \mathcal{L} \quad \text{sii} \quad C_{c\cdot n^c}(f'(x)) = 1 \quad \text{sii} \quad C_{c\cdot n^c}(C'_n(x)) = 1.$$

La composición[17] de los circuitos C y C' tiene tamaño polinomial, profundidad polilogarítmica, y se construye uniformemente a partir de 1^n. Esto prueba que $\mathcal{L}' \in \mathbf{NC}$. □

[16] Veamos que cualquier función h computable implícitamente en **L** es computable por una familia de circuitos **L**-uniforme de tamaño polinomial y profundidad polilogarítmica. Sea c tal que $|h(x)| \leq c\cdot|x|^c$ para todo $x \in \{0,1\}^*$. Definamos $\tilde{h}(x) \in \{0,1\}^{2\cdot c\cdot|x|^c}$ como

$$\tilde{h}(x) = h(x)(0)\ 0\ h(x)(1)\ 0 \ldots 0\ h(x)(|h(x)|-1)\ 1\ 0^{2c|x|^c - 2|h(x)|}$$

Observemos que $\tilde{h}(x)$ codifica $h(x)$ y su longitud con $2 \cdot c \cdot n^c$ bits, donde $n = |x|$. Definamos también para $i = 0, \ldots, 2 \cdot c \cdot n^c - 1$ el lenguaje

$$\mathcal{L}_{n,i} = \{\tilde{h}(x)(i) \in \{0,1\}^n : x \in \{0,1\}^n, \tilde{h}(x)(i) = 1\} \subseteq \{0,1\}^{2\cdot n^c}.$$

Es claro que $\mathcal{L}_{n,i} \in \mathbf{L}$. Por el Corolario 8, existen familias de circuitos $H_{n,i}$ **L**-uniformes que deciden $\mathcal{L}_{n,i}$ y son de tamaño polinomial en n y profundidad polilogarítmica en n. Más aun, los códigos de cada circuito $H_{n,i}$ se computan en espacio logarítmico por una máquina determinística uniformemente en n e i. Entonces, existe una máquina M tal que dado 1^n computa $c \cdot n^c$ en espacio logarítmico y luego computa uniformemente en i y n y usando de nuevo espacio logarítmico el código de un circuito $H_{n,i}$ de n entradas y una única salida que decide $\mathcal{L}_{n,i}$. Luego, M calcula el código de un circuito H_n con n entradas y $2 \cdot c \cdot n^c$ salidas que resulta de disponer todos los $H_{n,i}$ en paralelo, para $i = 0, \ldots, 2 \cdot c \cdot n^c - 1$. Este circuito H_n cumple que que para una entrada $x \in \{0,1\}^n$ devuelve $\tilde{h}(x) \in \{0,1\}^{2\cdot c\cdot n^c}$. M puede hacer esto en espacio logarítmico y uniformemente en n. Además, el circuito H_n que devuelve tiene tamaño polinomial, profundidad polilogarítmica y computa la función $\tilde{h}$.

[17] Siguiendo la nota 16, y tomando $h = f'$, la 'composición' se refiere a interpretar la salida de H_n para determinar

Referencias

[1] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009. https://theory.cs.princeton.edu/complexity/book.pdf.

[2] Lance Fortnow. A new proof of the nondeterministic time hierarchy, 2011. https://blog.computationalcomplexity.org/2011/04/new-proof-of-nondeterministic-time.html.

[3] Christos H Papadimitriou. *Computational complexity*. Pearson, 1993.

[4] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[5] Ingo Wegener. *Complexity theory: exploring the limits of efficient algorithms*. Springer Science & Business Media, 2005.

$h(x)$ con la información de $|h(x)|$ codificada en $\tilde{h}(x)$ y usarlo como entrada para $C_{|h(x)|}$. No es estrictamente una composición sino que requiere una lógica de circuitos intermedia y la aplicación de varios C_m en paralelo, para $m = 1, \ldots, c \cdot n^c$ (recordemos que $f'(|x|) \leq c \cdot |x|^c$). Todo esto se puede realizar dentro de **NC**, es decir, con familias de circuitos **L**-uniformes de tamaño polinomial y profundidad polilogarítmica.