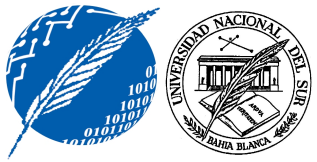
Algoritmos y Complejidad

Complejidad Computacional

Pablo R. Fillottrani

Depto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Primer Cuatrimestre 2017



Complejidad Computacional

Introducción

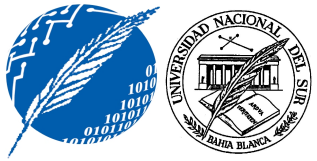
Clases de Complejidad

Clases **P** y **NP**

Problema **P** vs. **NP**

Otras clases de complejidad

Algoritmos de aproximación



Objetivos

- el concepto de correctitud de un algoritmo debe no sólo incluir el hecho de que el algoritmo realice lo especificado, sino también que lo realice con los recursos disponibles
- los recursos más limitados en la producción de software son:
  - el tiempo de ejecución (medido en cantidad instrucciones elementales)
  - la memoria para el almacenamiento de los datos (medida en general en cantidad de variables elementales).
- se denomina a estas funciones medidas de complejidad

- las medidas de complejidad interesantes no pueden ser estáticas (constantes para todas las instancias), sino dinámicas (dependen de la instancia a la cual se aplica el algoritmo)
- pero entonces no será factible estudiarlas para cada instancia en particular; se definen para todas las instancias de un determinado tamaño, en el peor caso
- el estudio de la eficiencia de los algoritmos no puede ser atacado por el argumento "las computadoras del futuro serán increíblemente rápidas y por lo tanto no interesará que el algoritmo sea eficiente"
- el ejemplo de los números de Fibonacci; o el hecho de que sin una mejora algorítmica resolver el problema del viajante para cien ciudades toma cien millones de años; o la factorización de un número de 300 dígitos que tarda un millón de años, refutan esta hipótesis

- para resolver estos problemas se necesita una mejora en los algoritmos; pero no una pequeña mejora sino una mejora en órdenes de magnitud
- por ejemplo, una búsqueda secuencial en un arreglo ordenado es de $\Theta(n)$ en el peor caso; una búsqueda binaria es de $\Theta(\log n)$. Para instancias pequeñas la diferencia puede no ser notable, pero para $n = 10^9$ la diferencia está entre esperar un año o un minuto por el resultado
- dado un algoritmo que resuelve un problema, es razonable (y aconsejable) preguntarse si no existirá un algoritmo más eficiente para el mismo problema
- el objetivo fundamental de la Complejidad Computacional es clasificar los problemas de acuerdo a su **tratabilidad**, tomando el o los algoritmos más eficientes para resolverlos

- esto es, poder determinar las respestuas a las siguientes preguntas:

  ¿Cuán tratable es el problema?
  Si el problema es tratable, ¿es el algoritmo suficientemente eficiente?

- si se demuestra que el problema no admite soluciones mejores, entonces se puede afirmar que el algoritmo es eficiente (salvo constantes ocultas!!)

Clasificación de problemas

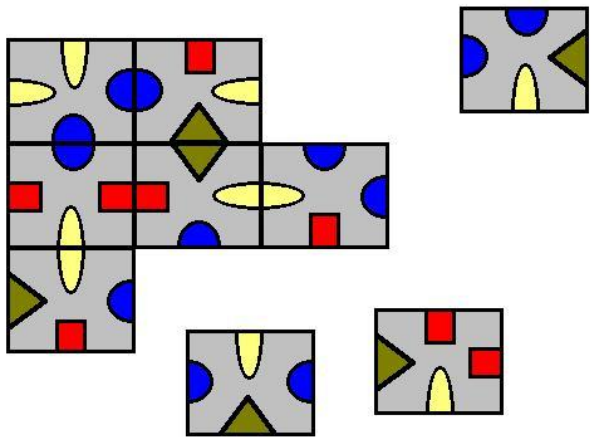
- de acuerdo al estado de conocimiento de sus algoritmos
  - cerrado
  - abierto
- de acuerdo a los recursos indispensables para su solución
  - tratables
  - intratables

- un problema se dice cerrado si se han encontrado algoritmos que lo resuelven y se ha demostrado que esos algoritmos son óptimos en cuanto al $\Theta()$ del tiempo de ejecución
- BÚSQUEDA en un arreglo ordenado y ORDENAMIENTO de un arreglo son problemas cerrados
- ÁRBOL DE CUBRIMIENTO MINIMAL para un grafo es un problema abierto, dado que su cota inferior demostrada es de $\Theta(a)$, mientras que el mejor algoritmo conocido no es lineal (pero mejor que $\Theta(a \log n)$)

- para cerrar un problema se puede hacer:
  - encontrar un algoritmo asintóticamente mejor que los que se conocen.
  - demostar una cota asintóticamente superior de las que se conocen.

- en general, los distintos grados de tratabilidad son muy subjetivos (varían mucho de acuerdo al modelo computacional, los recursos disponibles, las variantes de las estructuras de datos, etc.)
- por lo tanto un objetivo primario del estudio de la complejidad es definir cuáles problemas son tratables, y cuáles no. Recién después de esto se pueden considerar distintos grados de tratabilidad o intratabilidad
- por ejemplo, se puede afirmar que la mayoría de los problemas vistos en la materia son tratables: o sea tienen solución para instancias grandes, y una mejora algorítmica o una mejora en el HW produce una gran ampliación en el conjunto de instancias que se pueden resolver
- en cambio, hay problemas que no son tratables: el problema de las torres de Hanoi, o el problema del viajante, en la práctica sólo se resuelven para instancias pequeñas.

- no solo problemas con respuestas difíciles son intratables. Otro problema intratable es el denominado MONKEY PUZZLE, una especie particular de rompecabezas



- si se tiene un tablero de $n \times n$, no se conoce algoritmo mejor que probar todas las permutaciones posibles para resolverlo, esto es $\Omega(n!)$
- para $n = 5$, generando un millón de configuraciones por segundo, se tardaría 490.000 millones de años en resolverlo

- esta evidencia empírica ha permitido acordar (una especie de tesis de Turing-Church para la complejidad) que si un problema admite una solución polinomial entonces se trata de un problema **tratable**; en caso contrario se lo considera **intratable**
- el límite de tratabilidad entre algoritmos polinomiales y super-polinomiales es arbitrario, pero mayoritariamente aceptado. Por ejemplo, $1,00001^n$ o $n^{\log_2 n}$ serían intratables, mientras que n^{10000} o $100000000000n$ sería tratables
- el punto es que estas funciones extremas raramente aparecen en casos reales como tiempo de ejecución de algoritmos
- además, problemas como PROGRAMACIÓN LINEAL, con soluciones eficientes en general pero no polinomiales en el peor caso (Algoritmo Simplex), se demostraron tener solución polinomial.

- los algoritmos de $O(n^k)$ son considerados de tiempo razonable a pesar del k y de las posibles constantes ocultas
- para el espacio de memoria se tiene que tener más cuidado, porque ya un espacio polinomial muchas veces es intratable
- las funciones exponenciales y factoriales siempre son intratables
- además, dentro de esta tratabilidad y de esta intratabilidad es posible distinguir varios grados
- otro tema de investigación actual es la extensión del concepto de tratabilidad a algoritmos probabilísticos y paralelos. En estos casos se han propuesto otras formalizaciones

- para tener una mejor idea de la diferencia entre tratable e intratable:

n / funcion	10	50	100	300	1000
$5n$	50	250	500	1500	5000
$n \log_2 n$	33	282	665	2469	9966
n^2	100	2500	10000	90000	un millón (7 díg.)
n^3	1000	125000	un millón (7 díg.)	27 millones (8 díg.)	mil millones (10 díg.)
2^n	1024	16 díg.	31 díg.	91 díg.	302 díg.
$n!$	3600000 (7 díg.)	65 díg.	161 díg.	623 díg.	inimaginable
n^n	11 díg.	85 díg.	201 díg.	744 díg.	inimaginable

- para comparación: el número de protones en el universo es de 79 dígitos, y el número de microsegundos transcurridos desde el Big Bang es de 24 dígitos

- esto significa que si suponemos que se ejecutan k instrucciones por microsegundo, los algoritmos tardan

n / **funcion**	10	20	50	100	300
n^2	1/10000 segundo	1/2500 segundo	1/400 segundo	1/100 segundo	9/100 segundo
n^5	0,1 seg.	3,2 seg.	5,2 min.	2,8 hs.	28,1 días
2^n	1/1000 segundo	1 seg.	35,7 años	400 billones de siglos	— — —
n^n	2,8 hs.	3,3 billones años	— — —	— — —	— — —

- para comparación: se considera que el Big Bang ocurrió hace aproximadamente 15.000 millones de años

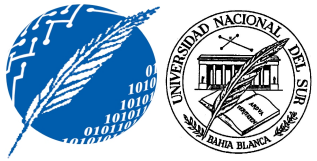
- supongamos que tenemos en la actualidad una computadora en la que ejecutamos distintos algoritmos, analizaremos cuál es la ventaja de adquirir una computadora más poderosa

tiempo algoritmo	instancia máxima posible de resolver con distintas computadoras		
	actual	100×	1000×
$\Theta(n)$	A	$100 \times A$	$1000 \times A$
$\Theta(n^2)$	B	$10 \times B$	$31,6 \times B$
$\Theta(2^n)$	C	$C+6,64$	$C+9,97$

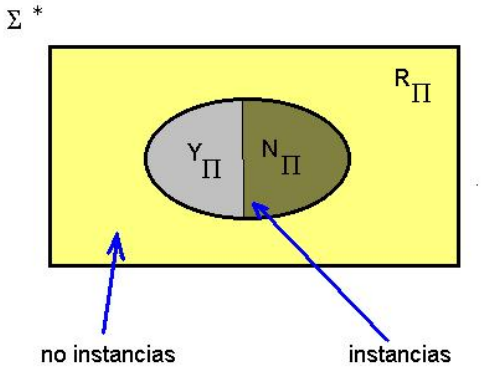
- esto demuestra que los problemas intratables no se resuelven comprando hardware; es una característica intrínsica del problema el hecho de que solo se resuelvan instancias pequeñas

Conceptos básicos

- con el objetivo de simplificar la clasificación, sólo se considerarán problemas de decisión, ie aquellos que tienen como respuesta **sí** o **no**
- los problemas que no son de decisión se analizan en forma separada, pero la mayoría se puede reducir a algún problema de decisión.
- a pesar de esta clasificación el espectro de problemas sigue siendo muy amplio. Se establecerá una asociación entre problemas de decisión y lenguajes formales, de forma de estudiar éstos últimos que son objetos más fáciles de tratar matemáticamente.



- sea Σ el alfabeto en el que estan codificadas las instancias a un determinado problema de decisión Π
- el conjunto Σ^* está particionado en tres clases: el conjunto Y_Π de las cadenas que codifican instancias cuya respuesta es **sí**; el conjunto N_Π de las cadenas que codifican instancias cuya respuesta es **no**; y R_Π el conjunto de cadenas que no codifican ninguna instacia



- se nota Y_Π como el lenguaje asociado a Π
- ejemplo: sea Π el problema de decidir si un número es primo, con las entradas dadas en binario; el lenguaje asociado es $Y_\Pi = \{\mathbf{10}, \mathbf{11}, \mathbf{101}, \mathbf{111}, \ldots\}$
- queda establecida la correspondencia

$$\text{Problemas de Decisión} \Longleftrightarrow \text{Lenguajes Formales}$$

- por lo tanto se clasificarán lenguajes formales, en lugar de problemas de decisión

- el alfabeto Σ de estos lenguajes no es importante para la clasificación, siempre y cuando contenga más de dos caracteres

Teorema 1

Sean Σ y Γ dos alfabetos tales que $\Gamma \geq 2$. Entonces existe una función $f : \Sigma^* \longrightarrow \Gamma^*$ biyectiva computable en tiempo lineal tal que $|f(x)| = \lceil \log_{|\Gamma|} |\Sigma| \rceil |x|$.

Demostración.

Se hace una codificación de todos los caracteres de Σ en secuencias de caracteres de Γ. □

- esto permite fijar como alfabeto estándar a $\Sigma = \{\mathbf{1}, \mathbf{0}, \sqcup\}$, cualquier lenguaje en otro alfabeto puede ser traducido a este en forma inmediata

- se usará como modelo de computación a las máquinas de Turing, con alfabeto Σ que siempre contiene el caracter en blanco $\sqcup$
- una máquina de Turing determinística de una cinta (MTD) es una tupla

$$\langle Q, \Sigma, \delta, q_0, q_1 \rangle$$

donde $q_0, q_1 \in Q$ y $\delta : Q \times \Sigma \longrightarrow Q \times \Sigma \times \{-1, 0, 1\}$

- una máquina de Turing no determinística de una cinta (MTND) es una tupla

$$\langle Q, \Sigma, \Delta, q_0, q_1 \rangle$$

donde $q_0, q_1 \in Q$ y Δ es un conjunto finito de tuplas de $Q \times \Sigma \times Q \times \Sigma \times \{-1, 0, 1\}$.

- sea T una MTD, y NT una MTND. La computación $T(x)$ se dice que acepta a x si $T(x)$ es finita y termina en el estado aceptador q_1. La computación $NT(x)$ se dice que acepta a x si existe en el árbol $T(x)$ un camino finito cuya hoja está en el estado aceptador q_1.
- el conjunto de entradas que T (o NT) acepta se denomina el lenguaje aceptado por T (o NT), y se nota $L(T)$ (o $L(NT)$)
- las propiedades siguientes ha sido vistas en materias previas

Teorema 2

Sea NT una MTND. Siempre existe una MTD T tal que $L(NT) = L(T)$ y si x es aceptada por NT en t pasos, entonces x es aceptada por T en c^t pasos, para alguna constante positiva c.

- un lenguaje $L \subseteq \Sigma^*$ es aceptable si existe una MTD T tal que $L = L(T)$
- un lenguaje $L \subseteq \Sigma^*$ es decidible si existe una MTD T tal que $L = L(T)$ y además, para todo $x \in \Sigma^*$ $T(x)$ es finita

Lema 3

Si un lenguaje L es decidible entonces L es aceptable.

Lema 4

Si un lenguaje L es decidible entonces L^C también es decidible.

Lema 5

Un lenguaje L es decidible si y solo si L y L^C son aceptables.

- también se demuestra que existen problemas decidibles, aceptables pero no decidibles y ni aceptables ni decidibles

- para comparar lenguajes, se usarán funciones de reducibilidad: dados dos lenguajes $A, B \subseteq \Sigma^*$, una función computable $f : A \longrightarrow B$ es una función de reducibilidad si para todo $x \in \Sigma^*$ vale $x \in A \leftrightarrow f(x) \in B$
- visto desde el punto de vista de los problemas de decisión, la función de reducibilidad es una transformación de instancias de un problema en instancias de otro problema tal que la solución al segundo es la solución al primero (una reducción entre problemas)
- las funciones de reducibilidad permiten definir una comparación entre lenguajes: se dice que $A \leq_m B$ si existe una función de reducibilidad de A hacia B
- la relación $\leq_m$ es una relación de pre-orden.

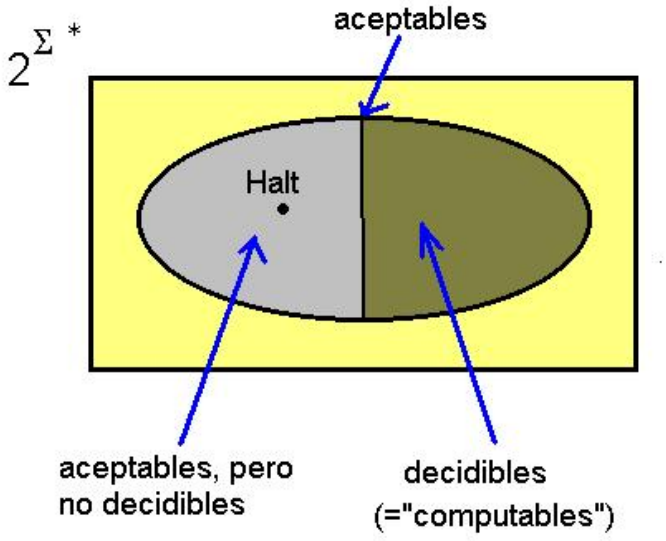
Lema 6

Si $L_1 \leq_m L_2$ y L_1 no es decidible, entonces L_2 no es decidible.

Lema 7

Sea $\emptyset \subset L_1 \subset \Sigma^*$. Entonces para todo lenguaje decidible L_2 vale que $L_2 \leq_m L_1$.

- las funciones de reducibilidad arbitrarias, junto con la comparación $\leq_m$, no son muy útiles porque pueden involucrar cantidades arbitrarias de recursos para ser computadas.
- en la práctica se definirán distintas restricciones (que la función sea computable en tiempo polinomial, o espacio logarítmico, etc.) para definir reducibilidades y comparaciones útiles entre problemas



- el estudio de la complejidad clasificará los lenguajes decidibles de acuerdo a la cantidad de recursos necesarios para su computación.

Clases de Complejidad

- si se quiere clasificar problemas, y se identifica a cada problema con un lenguaje formal, entonces es necesario estudiar clases de lenguajes
- una clase de lenguaje es un conjunto de lenguajes que cumple con alguna propiedad, es decir $\mathscr{C} = \{L : L \subseteq \Sigma^* \text{ y } \pi(L)\}$
- dadas dos clases de lenguajes $\mathscr{C}_1$ y $\mathscr{C}_2$, si se quiere mostrar que $\mathscr{C}_1 \subseteq \mathscr{C}_2$ entonces es suficiente con probar que para todo $L \subseteq \Sigma^*$ si vale $\pi_1(L)$ entonces también vale $\pi_2(L)$
- si además de $\mathscr{C}_1 \subseteq \mathscr{C}_2$ se quiere mostrar que la inclusión es estricta ($\mathscr{C}_1 \subset \mathscr{C}_2$), entonces se recurre a buscar un lenguaje separador que pertenezca a $\mathscr{C}_2$ pero no a $\mathscr{C}_1$.

- en complejidad, es un problema muy difícil encontrar lenguajes separadores
- muchas veces es útil la siguiente propiedad

Lema 8

Sean $\mathscr{C}_1 \subset \mathscr{C}_2$ dos clases de complejidad tal que $\mathscr{C}_1$ es cerrada c.r. a $\leq_r$. Entonces cualquier $L \in \mathscr{C}_2$ que sea $\mathscr{C}_2$-completo c.r. a $\leq_r$ es tal que $L \notin \mathscr{C}_1$.

Demostración.

Se supone por el absurdo que $L \in \mathscr{C}_1$, y se contradice el hecho de que $\mathscr{C}_1 \subset \mathscr{C}_2$. □

- una clase $\mathscr{C}$ es cerrada c.r. a $\leq_r$ si para todo $L_1 \in \mathscr{C}$ vale que $L_2 \leq_r L_1$ implica $L_2 \in \mathscr{C}$. Un lenguaje $L_1 \in \mathscr{C}$ es completo c.r. a $\leq_r$ si para todo $L_2 \in \mathscr{C}$ vale que $L_2 \leq_r L_1$
- entonces, si se dan las condiciones del lema pero no se sabe si $\mathscr{C}_1 \subseteq \mathscr{C}_2$ o $\mathscr{C}_1 \subset \mathscr{C}_2$, los lenguajes $\mathscr{C}_2$-completos son candidatos a lenguajes separadores

- las clases de lenguajes son conjuntos arbitrarios de lenguajes que no necesariamente comparten propiedades computacionales
- para que las clases de lenguajes sean clases de complejidad se deben definir los predicados $\pi(\cdot)$ en base a medidas de complejidad dinámicas
- sea $\{P_i\}$ un conjunto de algoritmos. Una medida de complejidad dinámica Φ es un conjunto de funciones Φ_i definidas sobre los naturales, que además cumplen con:
  - el dominio de P_i coincide con el de Φ_i.
  - para todo $i, x, m \in \mathbf{N}$ es computable si $\Phi_i(x) = m$.
- la primer condición impide mezclar lenguajes decidibles y no decidibles. La segunda condición es pide que sea computable la comparación del valor de la medida en cada instancia (y no que la función sea computable para todas las instancias)

Ejemplos

- sean $\{T_i\}$ el conjunto de todas las máquinas de Turing determinísticas
- se define *TIME* la medida $\Phi = \{TIME_i\}$, la cantidad de pasos de la MTD T_i en cada entrada x; entonces Φ es una medida de complejidad dinámica
- sea k la medida $\Phi = \{\mathbf{k}\}$ donde **k** es una función constante, k no cumple con la primer condición
- sea *VALOR* la medida $\Phi = \{VALOR_i\}$, el resultado que deja la MT T_i en la cinta al finalizar cada computación; *VALOR* no satisface la segunda restricción
- sea $SPACE^1$ la medida $\Phi = \{SPACE_i^1\}$, la cantidad de celdas que la MTD T_i usa en cada cada computación; $SPACE_1$ no satisface la primer restricción.

- sea $SPACE^2$ la medida $\Phi = \{SPACE_i^2\}$, la cantidad de celdas que la MTD T_i usa en cada computación que termina, o indefinido si no termina; $SPACE^2$ cumple la primer condición
- para la segunda se muestra

Lema 9

Sea T una MTD y x una entrada tal que $T(x)$ usa a lo sumo $h(|x|)$ celdas. Entonces o $T(x)$ para antes de $|\Sigma|^{h(|x|)}|Q|h(|x|)$ pasos, o $T(x)$ no termina.

Demostración.

Basta con considerar el conjunto de configuraciones globales posibles de la MTD T. Si pasan más de $|\Sigma|^{h(x)}|Q|h(x)$ configuraciones, entonces necesariamente alguna se repite y por lo tanto la máquina cicla indefinidamente.

- sean $\{T_i\}$ el conjunto de todas las máquinas de Turing no determinísticas
- sea *NTIME* la medida $\Phi = \{NTIME_i\}$, donde $NTIME_i$ es para cada $T_i(x)$
  1. el número de pasos del camino más corto que acepta $T_i(x)$ si existe al menos uno aceptador
  2. el número de pasos del camino más largo que rechaza $T_i(x)$ si todos los caminos terminan y son rechazadores
  3. indefinido en otro caso
- la medida *NTIME* así definida cumple con las condiciones de medida de complejidad dinámica
- la medida *NSPACE* se define en forma análoga, y también cumple con las condiciones.

- ahora sí se está en condiciones de definir clases de complejidad, para toda función $t : \mathbf{N} \longrightarrow \mathbf{N}$:

$$
\begin{aligned}
DTIME[t(n)] &= \{L : L \subseteq \Sigma^* \text{ y} \\
&\quad \exists T_i : L = L(T_i) \wedge TIME_i(x) \in O(t(|x|))\} \\
DSPACE[t(n)] &= \{L : L \subseteq \Sigma^* \text{ y} \\
&\quad \exists T_i : L = L(T_i) \wedge SPACE_i(x) \in O(t(|x|))\} \\
NTIME[t(n)] &= \{L : L \subseteq \Sigma^* \text{ y} \\
&\quad \exists NT_i : L = L(NT_i) \wedge NTIME_i(x) \in O(t(|x|))\} \\
NSPACE[t(n)] &= \{L : L \subseteq \Sigma^* \text{ y} \\
&\quad \exists NT_i : L = L(NT_i) \wedge NSPACE_i(x) \in O(t(|x|))\}
\end{aligned}
$$

- para que estas clases de complejidad tengan sentido (es decir, agrupen problemas con características computacionales parecidas) se restringen a funciones $t(x)$ denominadas tiempo-construíbles
- las funciones logarítmicas, polinomiales, exponenciales, factoriales son tiempo construíbles

- las clases de complejidad determinísticas satisfacen las siguientes propiedades

Lema 10

Para toda función tiempo-construíble t, siempre vale que $DTIME[t(n)] = coDTIME[t(n)]$.

Lema 11

Sea t una función tiempo-construíble tal que $t(n) \geq n$, y $L_1, L_2 \subset \Sigma^*$ tal que $|L_1 \triangle L_2| < \aleph_0$ (su diferencia es finita). Entonces $L_1 \in DTIME[t(n)]$ *si y solo si* $L_2 \in DTIME[t(n)]$.

- las clases de complejidad no determinísticas satisfacen la segunda propiedad, pero es un problema abierto determinar si satisfacen la primera

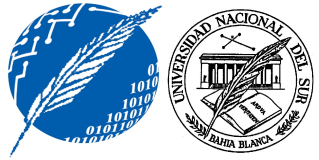
Clase **P**

- como nuestro objetivo es formalizar el concepto tratabilidad, y habíamos establecido que los algoritmos polinomiales constituyen un buena base para esto, entonces tiene sentido considerar a la siguiente clase:

$$\mathbf{P} = \cup_{k \in \mathbf{N}} DTIME[n^k]$$

- la mayoría de los problemas computacionales que se presentan en la práctica pertenecen a esta clase, como GCD, MULTIPLICACIÓN DE MATRICES, RECORRIDO DE UN GRAFO, CAMINO DE EULER, etc.

- en general se está de acuerdo que la clase **P** representa a los problemas tratables
- la mayoría de los problemas de esta clase tienen algoritmos con implementación razonable, a pesar de que el algoritmo polinomial para ese problema no sea el más conveniente de usar en la práctica (caso PROGRAMACIÓN LINEAL)
- sin embargo existen ciertos problemas en **P** que no son realmente tratables, debido a que:
  - el grado k del polinomio es un número grande
  - las constantes ocultas en la notación $O(\cdot)$ son altas
  - la demostración de pertenencia a **P** no es constructiva (es decir, no presenta un algoritmo).
- pero éstos son casos extremos, y se considera a **P** una razonable aproximación al concepto de tratabilidad



- otra característica interesante de la clase **P** es el hecho de que es robusta con respecto al modelo de computación
- esto significa que que la mayoría de los cambios de modelos y cambios de lenguajes conservan inalterable la clase
- por ejemplo, si se define la clase **P** en base a MTD de varias cintas, o en máquinas RAM, o en MTD con cintas infinitas en los dos extremos, etc, ya que existe una traducción polinomial entre cualquiera de estos modelos
- ejercicio: ¿qué implicaría tener definida una clase de problemas "tratables" que varíe con respecto al modelo de computación?)

- es posible demostrar que un problema pertenece a **P** mediante dos formas: mostrando un algoritmo polinomial, o usando una reducción a otro problema que ya se sabe que está en **P**
- el primer método es el que hemos usado para los ejemplos ya dados; para el segundo debemos definir reducibilidad polinomial: dados dos lenguajes L_1, L_2, L_1 se dice polinomialmente reducible a L_2 si existe un función f computable en tiempo polinomial tal que $x \in L_1 \leftrightarrow f(x) \in L_2$. Se nota $L_1 \leq_p L_2$.
- la reducción polinomial transforma en tiempo polinomial una instancia del problema L_1 en una instancia del problema L_2, de forma que sus respuestas sean las mismas
- la reducibilidad polinomial cumple con todas las propiedades generales de las funciones de reducibilidad vistas anteriormente

- además, permite afirmar que **P** es cerrada c.r. a $\leq_p$:

Teorema 12

Sean L_1, L_2 dos lenguajes tales que $L_1 \leq_p L_2$. Luego si $L_2 \in \mathbf{P}$ entonces $L_1 \in \mathbf{P}$.

Demostración.

Componiendo las MTD polinomiales que resuelven L_2 y que calculan la reducción, considerando que la composición de polinomios es un polinomio. □

Teorema 13

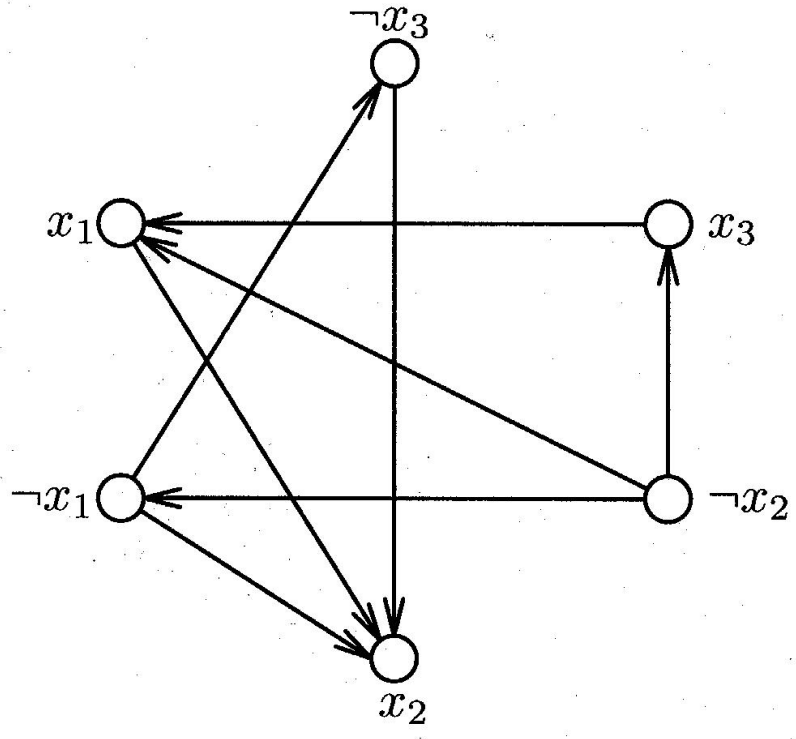
Sea L_1 un lenguaje tal que $\emptyset \subset L_1 \subset \Sigma^*$. Entonces para cualquier $L_2 \in \mathbf{P}$ vale que $L_2 \leq_p L_1$.

Demostración.

Si $\emptyset \subset L_1 \subset \Sigma^*$ entonces existe $z \in L_1$ y un $y \in \Sigma^* - L_1$. La reducción consiste para cada x resolver L_2 de acuerdo a su MTD polinomial, y asignarle z o y de acuerdo a si su respuesta es sí o no. □

- por ejemplo, sea 2-SAT el problema de decidir si una fórmula del cálculo proposicional con a lo sumo dos literales por clausula es satisfacible
- se mostrará que 2-SAT$\leq_p$CFC
- sea F una fbf de $\mathscr{L}$ con a los sumo dos literales por clausula. Se crea $G(F)$ con un nodo por cada posible literal, y un arco (L_1, L_2) si existe en F una clausula $\bar{L_1} \vee L_2$ o $L_2 \vee \bar{L_1}$ (o sea si $L_1 \rightarrow L_2$ o $L_2 \rightarrow L_1$)

- por ejemplo, para
  $F = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$, $G(F)$ es:



Teorema 14

Una fbf F es satisfacible si y solo si no existe una proposición p tal que p y $\neg p$ pertenecen al mismo CFC del grafo $G(F)$.

Demostración.

Sea F satisfacible. Si existen simultaneamente caminos de p a $\neg p$ y de $\neg p$ a p entonces $p \equiv \neg p$ es consecuencia de la formula, con lo que la formula es insatisfacible.
Sea $G(F)$ tal que ningún *CFC* contiene literales complementarios. Se construye una asignación de valores de verdad que hace a F satisfacible, asignando a todos los literales de un mismo *CFC* los mismos valores de verdad. Si existe camino de p a $\neg p$ entonces p es **falso**, los restante predicados se asignan arbitrariamente. □

- el teorema anterior, junto con el hecho de que $G(F)$ se puede calcular en tiempo polinomial permiten afirmar que 2-SAT$\leq_p$CFC
- además, como CFC$\in$ **P** entonces sabemos que 2-SAT$\in$ **P** ya que una combinación de algoritmos polinomiales resulta en un algoritmo polinomial (teorema 12)

- otro ejemplo de reducción es 2-COLOR$\leq_p$2-SAT
- 2-COLOR es el problema de, dado un grafo no dirigido G, encontrar una asignación de dos colores a todos los nodos de G (coloración) tal que ningún par de nodos adyacentes tengan el mismo color
- la reducción consiste en crear una proposición p_i por cada nodo n_i del grafo, que significará que el nodo i tiene asociado el primer color, mientras que su negación significa que tiene asociado el segundo color
- se crea entonces una fbf $F(G)$ con las clausulas $p_i \vee p_j$ y $\neg p_i \vee \neg p_j$ por cada arco (n_i, n_j) del grafo

Teorema 15

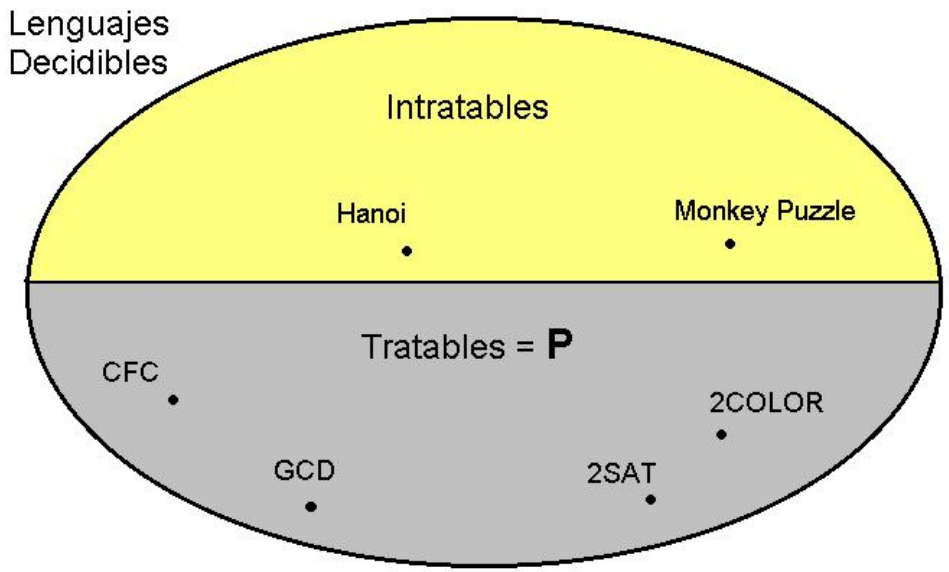
Un grafo no dirigido G tiene una 2-coloración si y solo si $F(G)$ es satisfacible.

Demostración.

$\Leftarrow$) Si G tiene una 2-coloración, entonces se crea una asignación de valores de verdad para cada p_i, arbitrariamente tomando un color como verdadero y otro como falso. Esta asignación satisface $F(G)$.
$\Rightarrow$) Si $F(G)$ es satisfacible, a partir de la asignación de valores de verdad se asigna un color a cada n_i tal que p_i es verdadero, y el otro color al resto. Esto forma una 2-coloración válida. □

- luego como la reducción es en tiempo polinomial, 2-SAT$\in$ **P** y **P** es cerrado c.r. a $\leq_p$ entonces 2-COLOR$\in$ **P**

- identificando la clase **P** con el concepto de tratabilidad, se estaría en las siguientes condiciones, con algunos problemas probadamente tratables y otros probadamente intratables:

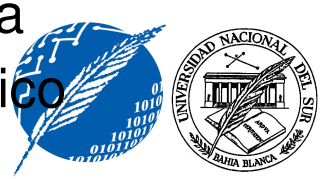


Clase **NP**

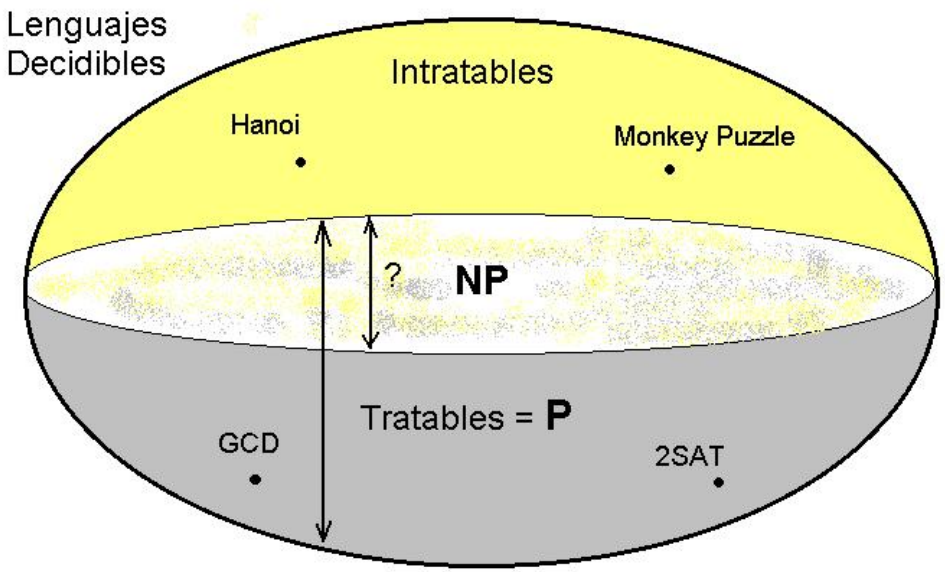
- en forma análoga a **P**, se puede definir su contraparte no determinística **NP**:

$$\mathbf{NP} = \cup_{k \in \mathbf{N}} NTIME[n^k]$$

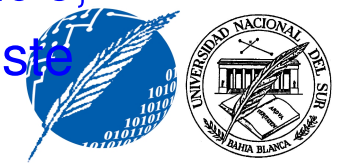
- por definición (toda computación determinística es trivialmente no determinística), sabemos que $\mathbf{P} \subseteq \mathbf{NP}$
- pero el problema de definir la "tratabilidad" no es tan sencillo: no se sabe si $\mathbf{P} = \mathbf{NP}$ o $\mathbf{P} \subset \mathbf{NP}$
- esto es, puede ser que todos los problemas en **NP** tengan solución determinística polinomial, o puede ser que no sea equivalente el tiempo determinístico con el no determinístico



- el teorema 2 solo establece la equivalencia en cuanto al poder computacional; pero el tiempo que usa es exponencial
- como consecuencia se tiene un conjunto de problemas cuya tratabilidad o no se desconoce:



- esta brecha es inaceptablemente muy grande, e impide de clasificar a importantes problemas como tratable o intratable
- desafortunadamente existe muchos problemas prácticos con estas características: tienen soluciones polinomiales no determinísticas, pero su máxima cota inferior es lineal
- por ejemplo:
  - EMPAQUETAMIENTO BIDIMENSIONAL
  - VIAJANTE
  - CIRCUITO HAMILTONIANO
  - MOCHILA con coeficientes enteros
  - SATISFACILIDAD en el cálculo de predicados
  - 3-COLOR de un grafo no dirigido
- para ninguno de ellos se ha encontado una solución tratable, pero tampoco se ha demostrado que esta solución no existe



- es importante conocer las propiedades de la clase **NP** para tratar de entender mejor el problema de su tratabilidad
- una característica interesante es que todos estos problemas parecen requerir, como parte de su naturaleza, construir soluciones parciales: empaquetamientos parciales, recorridos parciales, asignaciones de valores de verdad parciales, etc. Cuando una de estas soluciones parciales se detecta incorrecta, se debe realizar backtracking en busca de otra alternativa
- por lo tanto es difícil decidir si existe o no una solución; es decir dar la respuesta **sí** o **no**
- en cambio es interesente notar que en el caso que la respuesta sea **sí** es fácil convencer a alguien de ello. Esto es lo que se denomina certificado, cuyo tamaño siempre puede ser acotado por $\Theta(n)$

- existe otra forma de ver este hecho: supongamos que disponemos de una moneda mágica que se usa en el proceso de bactracking para determinar en cada decisión cuál es el camino correcto
- la moneda siempre “adivina” la solución. En este caso la el algoritmo es polinomial

Teorema 16

Un lenguaje L pertenece a **NP** *si y solo si existe un lenguaje* $L_{check} \in \mathbf{P}$ *tal que*

$$L = \{x : \exists y \langle x, y \rangle \in L_{check} \; y \; |y| \leq p(|x|)\}$$

siendo p un polinomio.

Demostración.

Transformando una máquina de Turing. □

- este teorema describe una forma alternativa de probar la pertenencia a **NP**: mostrar que su lenguaje L_{check} pertenece a **P**

- por ejemplo
  - para mostrar que SAT$\in$ **NP** es suficiente con dar un algoritmo determinístico polinomial para chequear si una asignación de valores de verdad hace verdadera una fbf.
  - para mostrar que la versión de decisión del problema VIAJANTE (dado un grafo y un k encontrar un circuito en el grafo de longitud menor o igual a k) pertenece a **NP** se da un algoritmo polinomial para controlar si un dado circuito es un circuito de viajante con longitud menor o igual a k
  - se puede mostrar que 3-COLOR$\in$ **NP** dando un algoritmo polinomial para controlar que una asignación de colores de un grafo asigne colores diferentes a nodos adyacentes.

Clase **NPC**

- como **P** es cerrada c.r. a $\leq_p$, el lema 8 permite suponer que los lenguajes **NP**-completos c.r. a $\leq_p$ (**NPC**) son candidatos a lenguajes separadores, es decir a pertenecer a $\mathbf{NP} - \mathbf{P}$
- entonces, para averiguar si $\mathbf{P} = \mathbf{NP}$ es suficiente con establecer el status de un problema en **NPC**: si uno de estos problemas tiene solución polinomial, entonces todos los **NP** tienen solución polinomial; si se demuestra que no puede existir una solución polinomial para alguno de ellos, entonces ninguno de **NPC** la tiene
- el destino de un problema **NPC** es el destino de todos: todos son tratables o todos son intratables.
- por lo tanto es interesante conocer cuáles problemas pertenecen a **NPC**

Teorema de Cook

Teorema 17 (Cook/Levin)

SAT es un problema en **NPC**.

Demostración.

Primero se prueba que SAT$\in$ **NP** (usando el teorema 16 o mostrando un algoritmo no determinístico polinomial). Luego se toma un lenguaje arbitrario $L \in$ **NP**, y se muestra que $L \leq_p$SAT. Para esto se define primero la reducción: sea $T = \langle Q, \Sigma, \Delta, q_0, q_1 \rangle$ la MTND que acepta a L, y $x \in \Sigma^*$. Como $L \in$ **NP** sabemos que $T(x)$ para (aceptando o rechazando) en tiempo polinomial $p(|x|)$.
A continuación se define una fbf $F(T, x)$ tal que $F(T, x)$ sea satisfacible si y solo si $T(x)$ termina en el estado aceptador.

Demostración (cont.)

La fórmula $F(T,x)$ está formada por las siguientes proposiciones:

- para cada paso $t, 0 \leq t \leq p(|x|)$, para cada estado $q_e, 1 \leq e \leq |Q|$, existe Q_e^t para representar que en el paso t la computación $T(x)$ está en el estado e
- para cada paso $t, 0 \leq t \leq p(|x|)$, para cada celda $c_s, 1 \leq s \leq p(|x|)$, existe L_s^t para representar que en el paso t la computación $T(x)$ tiene la cabeza lectora en la posición s.
- para cada paso $t, 0 \leq t \leq p(|x|)$, para cada celda $c_s, 1 \leq s \leq p(|x|)$, para cada letra $l_i \in \Sigma \cup \{\sqcup\}$ (o sea $l_0 = 0, l_1 = 1, l_2 = \sqcup$), existe $P_{s,i}^t$ para representar que en el paso t la computación $T(x)$ tiene en la celda c_s la letra l_i.

Demostración (cont.)

A partir de estas proposiciones, se define

$$F(T,x) = A \wedge B \wedge C \wedge D \wedge E \wedge F$$

donde A, B, C describen el comportamiento general de las MTND; D describe la configuración inicial de $T(x)$; E describe el comportamiento particular de T determinado por sus tuplas Δ; y F describe la configuración aceptadora de $T(x)$. □

Demostración (cont.)

Las subfórmulas A, B, C tienen la siguiente forma:

$$A = \bigwedge_{0 \le t \le p(|x|)} [(\bigvee_{1 \le e \le |Q|} Q_e^t) \wedge \bigwedge_{\substack{1 \le e,e' \le |Q| \\ e \neq e'}} (Q_e^t \rightarrow \neg Q_{e'}^t)]$$

$$B = \bigwedge_{0 \le t \le p(|x|)} [(\bigvee_{1 \le s \le p(|x|} L_s^t) \wedge \bigwedge_{\substack{1 \le s,s' \le p(|x|) \\ s \neq s'}} (L_s^t \rightarrow \neg L_{s'}^t)]$$

$$C = \bigwedge_{\substack{0 \le t \le p(|x|) \\ 1 \le s \le p(|x|)}} [(P_{s,0}^t \vee P_{s,1}^t \vee P_{s,2}^t) \wedge (\bigwedge_{\substack{1 \le i,i' \le 2 \\ i \neq i'}} (P_{s,i}^t \rightarrow \neg P_{s,i'}^t)]$$

Demostración (cont.)

Las subfórmulas D, E, F tienen la siguiente forma:

$$D = \Big(\bigwedge_{1\le j\le |x|} P^0_{j,i_j}\Big) \wedge \Big(\bigwedge_{|x|<j\le p(|x|)} P^0_{j,2}\Big) \wedge Q^0_0 \wedge L^0_1$$

$$E = \bigwedge_{\substack{0\le t\le p(|x|)\\ 1\le s\le p(|x|)}} \Big[\bigvee_{\langle q_i,p_a,q_j,p_b,m\rangle\in\Delta} (Q^t_i \wedge L^t_s \wedge P^t_{s,a} \to Q^{t+1}_j \wedge L^{t+1}_{s+m} \wedge P^{t+1}_{s,b})\Big]$$

$$F = \bigvee_{1\le t\le p(|x|)} Q^t_1$$

siendo $x = l_{i_1} l_{i_2} \ldots l_{i_{|x|}}$ con $l_{i_k} \in \Sigma$. □

Demostración (cont.)

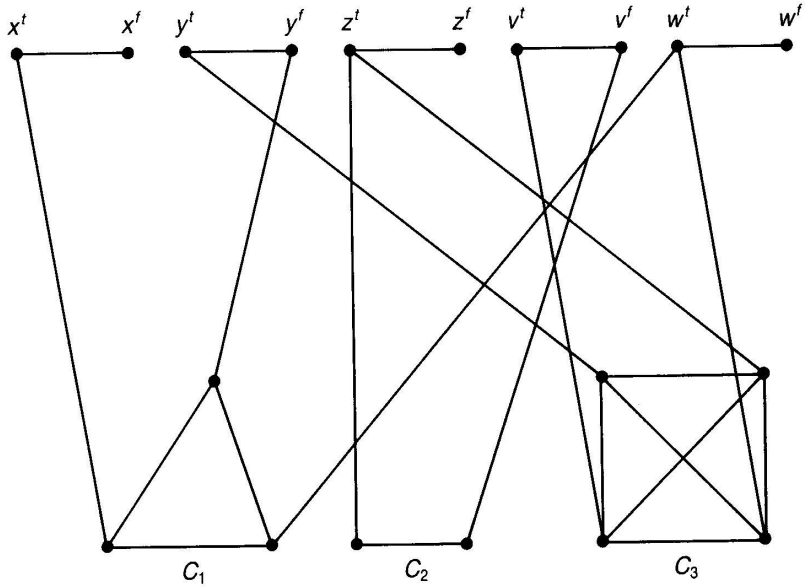
Se puede mostrar que $F(T,x)$ tiene una longitud polinomial en $|x|$, ya que la su longitud es cúbica en $p(|x|)$.
Además, vale que si $F(T,x)$ es satisfacible entonces $T(x)$ acepta a x y que si $T(x)$ acepta a x entonces $F(T,x)$ es satisfacible. (Los detalles de estos dos últimos puntos queda como ejercicio.)
Esto finaliza la prueba que $L \leq_p \text{SAT}$, y como $L \in \mathbf{NP}$ es arbitrario, entonces SAT es *NP*-completo. □

- la importancia del teorema de Cook radica en que a partir del conocimiento de que SAT$\in$ **NPC**, es mucho más fácil probar que otros problemas son **NPC** a través de la reducción polinomial: $L \in$ **NPC** si y solo $L_2 \in$ **NPC**, $L \in$ **NP** y $L_2 \leq_p L$.
- de esta forma más de 1000 problemas de dominios muy diferentes, y con variadas aplicaciones, se han probado que pertenecen a **NPC**
- el resultado es muy frustrante, ya que para ninguno de estos problemas se ha podido encontrar una solución polinomial, y tampoco se les ha podido demostrar una cota inferior mayor que lineal.

Reducción SAT$\leq_p$CUBR.NODOS

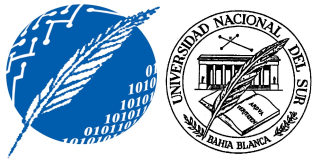
- sea F una fbf en FNC, se debe especificar un grafo G y un número K tal que F es satisfacible si y solo si G tiene un K-cubrimiento de nodos
- sean $p_1, \ldots, p_n$ las proposiciones de F, y $C_1, \ldots, C_m$ sus clausulas. $G(F)$ se construye:
  1. por cada p_i se agregan dos nodos en G etiquetados p_i y $\neg p_i$. Cada par de estos nodos se conecta con un arco en G
  2. por cada C_j con n_j literales, se agrega en G un subgrafo completo de n_j nodos etiquetados con cada literal de C_j
  3. se unen con un arco cada nodo agregado por la segunda regla con el nodo agregado en la primera que tiene igual etiqueta. El K se determina por la siguiente fórmula $K = n + \sum_{j=1}^{m}(n_j - 1)$.

- efectivamente, se muestra que F es satifacible si y solo si $G(F)$ tiene un K-cubrimiento de nodos, que la reducción es en tiempo polinomial, y que CUBR.NODOS$\in$ **NP**, con lo que se concluye que CUBR.NODOS$\in$ **NPC** (ejercicio)
- por ejemplo, para $F = (x \vee \neg y \vee w) \wedge (z \vee \neg v) \wedge (v \vee y \vee z \vee w)$ se tiene el siguiente grafo $G(F)$, con $K = 11$:



Reducción SAT$\leq_p$3-SAT

- sea F una fbf en CNF, se debe especificar una fbf F' una fbf en CNF con exactamente 3 literales por clausula tal que F es satisfacible si y solo si F' es satisfacible
- la fbf F' se construye a partir de cada clausula C_i de F de la siguiente forma:
  - si $C_i = L_{i_1}$ entonces $C'_i = L_{i_1} \vee L_{i_1} \vee L_{i_1}$.
  - si $C_i = L_{i_1} \vee L_{i_2}$ entonces $C'_i = L_{i_1} \vee L_{i_2} \vee L_{i_2}$.
  - si $C_i = L_{i_1} \vee L_{i_2} \vee L_{i_3}$ entonces $C'_i = C_i$.
  - si $C_i = L_{i_1} \vee \ldots \vee L_{i_k}$, $k > 3$ entonces $C'_i = (L_{i_1} \vee L_{i_2} \vee y_{i_1}) \wedge (\neg y_{i_1} \vee L_{i_3} \vee y_{i_2}) \vee \ldots \vee (\neg y_{i_{k-4}} \vee L_{i_{k-2}} \vee y_{i_{k-3}}) \vee (\neg y_{i_{k-3}} \vee L_{i_{k-1}} \vee L_{i_k})$, siendo $y_{i_j}, 1 \leq j < k-2$ nuevas letras proposicionales.
- es posible mostrar también que la reducción se puede hacer en tiempo polinomial, y como también 3-SAT$\in$ **NP** entonces 3-SAT$\in$ **NPC**.



- por ejemplo, si

$$F = (p \vee \neg q \vee \neg r \vee \neg s \vee w \vee \neg v) \wedge (\neg w \vee z)$$

entonces

$$\begin{aligned} F' \quad = \quad & (p \vee \neg q \vee y_{11}) \wedge (\neg y_{11} \vee \neg r \vee y_{12}) \wedge \\ & \wedge (\neg y_{12} \vee \neg s \vee y_{13}) \wedge (\neg y_{13} \vee w \vee \neg v) \wedge (\neg w \vee z \vee z) \end{aligned}$$

- la satisfacibilidad de la clausula original garantiza la satisfacibilidad de las clausulas transformadas asignando **verdadero** a todas las y_{i_j} hasta encontrar el literal verdadero de la clausula, luego los siguiente y_{i_k} toman valor **falso**

- algunos problemas de **NPC** parecen similares a otros que se saben en **P**, pero tiene complejidad distinta

Problema en **P**	Problema en **NPC**
CUBRIMIENTO DE ARCOS Instancia: grafo $G = \langle N, A \rangle$, entero positivo k. Respuesta: **sí** si existe $A' \subseteq A$ tal que $\lvert A' \rvert \leq k$ y que todo nodo de N es el extremo de algún arco en A'.	CUBRIMIENTO DE NODOS Instancia: grafo $G = \langle N, A \rangle$, entero positivo k. Respuesta: **sí** existe $N' \subseteq N$ tal que $\lvert N' \rvert \leq k$ y que todo arco de A toca N'.

Problema en **P**	Problema en **NPC**
CONJUNTO DE ARCOS NO DIRIGIDOS RETROALIMENTADOS Instancia: grafo $G = \langle N, A \rangle$, entero positivo k. Respuesta: **sí** si existe un subconjunto $A' \subseteq A$ tal que $\lvert A' \rvert \leq k$ y que todo ciclo en G contiene un arco de A'.	CONJUNTO DE ARCOS DIRIGIDOS RETROALIMENTADOS Instancia: grafo dirigido $G = \langle N, A \rangle$, entero positivo k. Respuesta: **sí** si existe un subconjunto $A' \subseteq A$ tal que $\lvert A' \rvert \leq k$ y que todo ciclo (dirigido) en G contiene un arco de A'.

Problema en **P**	Problema en **NPC**
CICLO DE EULER Instancia: grafo $G = \langle N, A \rangle$, $\lvert A \rvert = a$. Respuesta: **sí** si existe un orden $(n_i, m_i), 1 \leq i \leq a$ de los arcos de manera que $m_i = n_{i+1}, 1 \leq i < m$ y $n_a = m_1$.	CICLO DE HAMILTON Instancia: grafo $G = \langle N, A \rangle$, $\lvert N \rvert = n$. Respuesta: **sí** si existe un orden $n_i, 1 \leq i \leq n$ de los nodos de manera que $(n_i, n_{i+1}) \in A, 1 \leq i < n$ y $(n_n, n_1) \in A$.

Problema en **P**	Problema en **NPC**
2-SATISFACIBILIDAD Instancia: fbf F del cálculo proposicional en FNC de manera que ninguna clausula contenga más de 2 literales. Respuesta: **sí** si F es satisfacible.	3-SATISFACIBILIDAD Instancia: fbf F del cálculo proposicional en FNC de manera que ninguna clausula contenga más de 3 literales. Respuesta: **sí** si F es satisfacible.

Problema en **P**	Problema en **NPC**
ECUACIÓN DIOFANTINA LINEAL Instancia: enteros positivos a, b, c. Respuesta: **sí** si existen enteros positivos x, y tales que $ax + by = c$.	ECUACIÓN DIOFANTINA CUADRÁTICA Instancia: enteros positivos a, b, c. Respuesta: **sí** si existen enteros positivos x, y tales que $ax^2 + by = c$.

Problema en **P**	Problema en **NPC**
PARTICIONES UNARIAS Instancia: conjunto $A = \{a_i\}$ de enteros en notación unaria. Respuesta: **sí** si existe un subconjunto $A' \subset A$ tal que $\sum_{a_i \in A'} a_i = \sum_{a_i \in A - A'} a_i$.	PARTICIONES BINARIAS Instancia: conjunto $A = \{a_i\}$ de enteros en notación binaria. Respuesta: **sí** si existe un subconjunto $A' \subset A$ tal que $\sum_{a_i \in A'} a_i = \sum_{a_i \in A - A'} a_i$.

Problemas en **P**	Problemas en **NPC**
CAMINO MÁS CORTO Instancia: grafo $G = \langle N, A \rangle$, función distancia $d : A \to \mathbf{N}$, nodos $a, b \in N$, entero positivo k. Respuesta: **sí** si existe en G un camino simple de a a b de longitud menor o igual a k.	CAMINO MÁS LARGO Instancia: grafo $G = \langle N, A \rangle$, función distancia $d : A \to \mathbf{N}$, nodos $a, b \in N$, entero positivo k. Respuesta: **sí** si existe en G un camino simple de a a b de longitud mayor o igual a k.

Problema en **P**	Problema en **NPC**
INTREE SCHEDULING Instancia: conjunto T de tareas de duración unitaria, $p : T \to \mathbf{N}$ plazo máximo de cada tarea, orden parcial $<$ en T tal que toda tarea tenga a lo sumo un sucesor inmediato, entero positivo m. Respuesta: **sí** si existe un scheduling de T en m procesadores que obedezca el orden $<$ y ejecute $t \in T$ antes de su plazo $p(t)$.	OUTTREE SCHEDULING Instancia: conjunto T de tareas de duración 1, $p : T \to \mathbf{N}$ plazo máximo de cada tarea, orden parcial $<$ en T tal que toda tarea tenga a lo sumo un predecesor inmediato, entero positivo m. Respuesta: **sí** si existe un scheduling de T en m procesadores que obedezca el orden $<$ y ejecute $t \in T$ antes de su plazo $p(t)$.

- después de más de tres décadas de estudio, los investigadores no han podido responder la pregunta $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$
- es decir, no han encontrado un algoritmo polinomial ni han demostrado una cota superpolinomial, para ningún problema de **NPC**
- sin embargo, la experiencia práctica indica que $\mathbf{P} \subset \mathbf{NP}$ es más plausible que $\mathbf{P} = \mathbf{NP}$. Se cree que años de búsqueda de algoritmos más eficientes hacen más plausible la idea de que no existan
- esta experiencia también prefiere la idea de que una resolución de este problema (en uno u otro sentido) provendrá indirectamente del estudio detallado de las propiedades de las clases, y no a través de una prueba directa.

- por ejemplo, se ha mostrado que:

Teorema 18

*Si todo par de lenguajes $L_1, L_2 \in$ **NPC** es tal que L_1 y L_2 son p-isomorfos, entonces* $\mathbf{P} \neq \mathbf{NP}$.

donde dos lenguajes son *p*-isomorfos si son reducibles entre sí por una función de reducibilidad polinomial, biyectiva y con inversa también polinomial

- y en el otro sentido

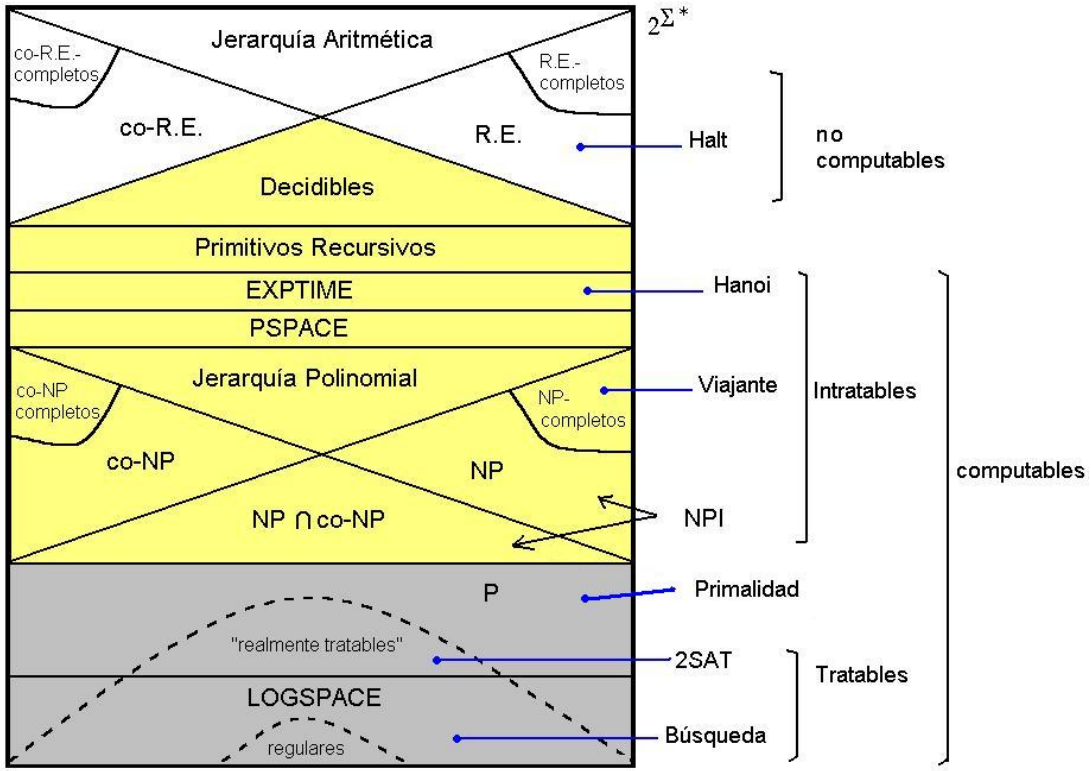
Teorema 19

*Si existe un lenguaje $L \in$ **NPC** tal que L es ralo, entonces* $\mathbf{P} = \mathbf{NP}$.

donde un lenguaje es ralo si su función censo está acotada por un polinomio

- existen muchos otros resultados como los anteriores, que favorecen una postura u otra, pero todas las condiciones que estos resultados establecen paracen ser tanto o más esquivas que el problema original
- en el caso que sea $\mathbf{P} \neq \mathbf{NP}$, se puede mostrar que existen lenguajes en $\mathbf{NP} - \mathbf{P}$ que no son **NPC**
- se los denomina **NP**-intermedios, o **NPI**. Se supone que varios problemas pertenecen a esta clase

Famila de clases de complejidad



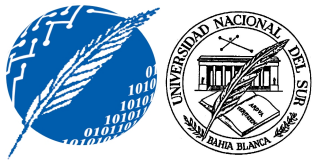
siempre que **P** $\neq$ **NP**

Clases de Complejidad en Espacio

- análogamente a **P** y **NP**, se pueden definir clases de complejidad polinomial en espacio determinística y no determinística

$$\mathbf{PSPACE} = \cup_{k \in \mathbf{N}} DSPACE[n^k]$$

$$\mathbf{NPSPACE} = \cup_{k \in \mathbf{N}} NSPACE[n^k]$$



- uno de los resultados más tempranos en cuanto al estudio de la complejidad establece que la computación no determinística en espacio es sorprendentemente eficiente comparando con la computación determinística equivalente

Teorema 20 (Savitch)

Para toda función $f : \mathbf{N} \longrightarrow \mathbf{N}$ *tal que* $f(n) \geq \log n$ *vale que*

$$NSPACE[f(n)] \subseteq DSPACE[f(n)^2]$$

Corolario 21

PSPACE $=$ **NPSPACE**

Demostración.

Por el teorema de Savitch, $NSPACE[n^k] \subseteq DSPACE[n^{k^2}]$.

- también sabemos que $\mathbf{P} \subseteq \mathbf{PSPACE}$ y que $\mathbf{NP} \subseteq \mathbf{NPSPACE}$, luego se tienen las siguientes relaciones

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME} = \cup_{k \in \mathbf{N}} DTIME[2^{n^k}]$$

- no se conoce si algunas de estas inclusiones son realmente igualdades

- sin embargo, sí se sabe que $\mathbf{P} \subset \mathbf{EXPTIME}$, por lo que necesariamente por lo menos una inclusión es estricta

Teorema 22 (Teorema de Jerarquía en Tiempo)

Para toda función tiempo construible $t : \mathbf{N} \longrightarrow \mathbf{N}$ *existe un lenguaje* L *tal que* L *es decidible en tiempo* $O(t(n))$ *pero no en tiempo de* $o(t(n)/\log n)$.

Corolario 23

Para todo par de números reales $\varepsilon_1, \varepsilon_2$ *tal que* $1 \leq \varepsilon_1 < \varepsilon_2$ *vale que* $DTIME[n^{\varepsilon_1}] \subset DTIME[n^{\varepsilon_2}]$.

Corolario 24

$\mathbf{P} \subset \mathbf{EXPTIME}$

- también es posible demostrar un teorema equivalente para la jerarquía en espacio, donde una función es espacio construible si cumple la definición análoga a tiempo construible pero con restricción en el espacio

Teorema 25 (Teorema de Jerarquía en Espacio)

Para toda función espacio construible $t : \mathbf{N} \longrightarrow \mathbf{N}$ *existe un lenguaje* L *tal que* L *es decidible en tiempo* $O(t(n))$ *pero no en tiempo de* $o(t(n))$.

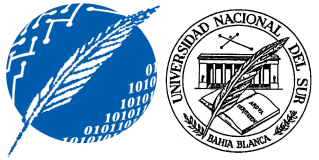
Corolario 26

Para todo par de números reales $\varepsilon_1, \varepsilon_2$ *tal que* $1 \leq \varepsilon_1 < \varepsilon_2$ *vale que* $DSPACE[n^{\varepsilon_1}] \subset DSPACE[n^{\varepsilon_2}]$.

- existen problemas **PSPACE**-completos, que según el lema anterior serían candidatos a lenguajes separadores en **PSPACE** y todas las clases inferiores
- igualmente, no se sabe si estos problemas pertenecen a **P** o no.
- son problemas **PSPACE**-completos:
  - TQBF el problema de determinar si una fórmula booleana totalmente cuantificada es verdadera o falsa
  - saber si un jugador tiene una estrategia ganadora en un grafo generado por el juego de GEOGRAFIA GENERALIZADO.

Clases **L** y **NL**

- es posible definir relaciones de reducibilidad de granularidad más fina que la reducibilidad polinomial, en particular para estudiar la complejidad en espacio logarítmico
- la reducibilidad logarítmica se define: dados dos lenguajes L_1, L_2, L_1 se dice logarítmicamente reducible a L_2 si existe un función f computable en espacio logarítmico tal que $x \in L_1 \leftrightarrow f(x) \in L_2$, y se nota $L_1 \leq_l L_2$
- la reducibilidad logarítmica cumple con todas las propiedades generales de las funciones de reducibilidad vistas anteriormente
- de esta forma se pueden definir clases y lenguajes completos dentro de la clase **P**



- las clases **L** y **NL** contienen los problemas solubles determinísticamente y no determinísticamente en espacio logarítmico

$$\mathbf{L} = DSPACE[\log n]$$

$$\mathbf{NL} = NSPACE[\log n]$$

- por supuesto $\mathbf{L} \subseteq \mathbf{NL}$, pero al igual que en el tiempo polinomial, no se sabe si la inclusión es estricta

- el problema CAMINO para determinar si existe un camino en un grafo dirigido entre un dado par de nodos (que se puede resolver con el algoritmo de Dijkstra) se puede demostrar que es **NL**-completo con respecto a $\leq_l$
- dado que CAMINO tiene solución determinística polinomial en tiempo, entonces como consecuencia se tiene que **NL** $\subseteq$ **P**
- otro resultado sorprendente es que a pesar que no se sabe si **NP** $=$ *co***NP**, se puede demostrar que **NL** $=$ *co***NL**

- en resumen, sabemos que

$$\mathbf{L} \subseteq \mathbf{NL} = co\mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{PSPACE}$$

- y por el teorema de la jerarquia en espacio se sabe que $\mathbf{L} \subset \mathbf{PSPACE}$, por lo que también alguna de estas inclusiones necesariamente debe ser estricta, sin saber cuál
- la clase en tiempo logarítmico **LOGTIME** no constituye una clase de problemas interesante ya que son problemas que se resuelven sin ni siquiera leer totalmente la entrada. A esta clase pertenece BÚSQUEDA BINARIA

Otras Clases de Complejidad

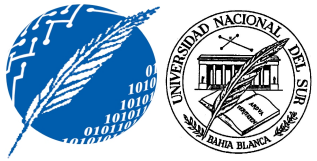
- clases probabilísticas
- jerarquía polinomial en tiempo
- clases de computación paralela
- jerarquía aritmética de incomputabilidad

Algoritmos de aproximación

- el hecho de conocer que un problema es **NP**-completo es instructivo
- puede ser útil para no perder tiempo en búsqueda de un algoritmo eficiente que probablemente no exista
- sin embargo, esto no significa que el problema no se pueda solucionar

- un algoritmo heurístico, o simplemente heurística, es un procedimiento que eventualmente puede producir una solución buena, o incluso optimal, a un problema; pero también puede producir una mala solución o incluso ninguna
- la heurística puede ser determinística o probabilística. La principal diferencia con los algoritmos Monte Carlo es que no está garantizada una baja probabilidad de error
- un algoritmo de aproximación es un procedimiento que siempre resulta en una solución al problema, aunque puede fallar en encontrar la solución optimal
- para que sea útil, debe ser posible calcular una cota o sobre la diferencia, o sobre el cociente entre la solución optimal y la devuelta por el algoritmo.

- consideremos por ejemplo el problema de la mochila, con coeficientes enteros
- la solución greedy se mostró no optimal, mientras que la solución de programación dinámica era de $\Theta(nW)$ que puede ser muy mala si W es grande
- no es un problema en **P**
- la solución *greedy* se puede extender fácilmente a un algoritmo de aproximación que garantice una solución por lo menos la mitad que la solución optimal
- el algoritmo de aproximación supone que ningún objeto pesa más que la capacidad de la mochila, pero esta restricción es fácil de levantar



Algoritmo *greedy*

```
PROCEDURE MochilaG(w[1..n],v[1..n],W)
 ordenar v[i]/w[i] en L
 peso::=0; valor::=0
 FOR i::=1 TO n
   IF peso+w[i]<=W
     valor::=valor+v[i]
     peso::=peso+w[i]
   ENDIF
 ENDFOR
 RETURN valor
```

Algoritmo de aproximación

```
PROCEDURE MochilaA(w[1..n],v[1..n],W)
  maxElto::=max{v[i], 1<=i<=n}
  greedy::=MochilaG(w,v,W)
  RETURN max{maxElto, greedy}
```

- sean opt el valor de la carga óptima, y $\widetilde{opt}$ el valor retornado por el algoritmo de aproximación.

Teorema 27

Vale que $\widetilde{opt} \geq opt/2$.

Demostración.

Si todos los objetos caben en la mochila, entonces $\widetilde{opt} = opt$. Sino, existe k el menor índice tal que $\sum_{i=1}^{k} w[i] > W$. Sea opt' el valor de una carga óptima con $W' = \sum_{i=1}^{k} w[i]$. Vale que $opt' = \sum_{i=1}^{k} v[i]$ y como $W' > W$, $opt' > opt$. Luego

$$\begin{aligned} \widetilde{opt} &= \textbf{max}\{\texttt{maxElto}, \texttt{greedy}\} \geq (\texttt{maxElto} + \texttt{greedy}))/2 \\ &\geq (v[k] + \sum_{i=1}^{k-1} v[i])/2 = (\sum_{i=1}^{k} v[i])/2 = opt'/2 \geq opt/2 \end{aligned}$$

- se conocen mejores algoritmos de aproximación para este problema, algunos basados incluso en la solución de programación dinámica
- otros algoritmos de aproximación trabajan sobre determinados subproblemas del problema original
- como la mayoría de los problemas **NPC** tienen una gran cantidad de instancias a las cuales se aplican, se puede llegar a tener suerte caracterizando un subconjunto de instancias para las cuales existe un algoritmo de aproximación eficiente y confiable
- los algoritmos de aproximación también son aplicables en ocasiones donde los datos iniciales son inciertos
- en estos casos, aún un algoritmo exacto puede dar una solución mala debido a la imprecisión de los datos originales.