Introducción a la teoría de la $\mathcal{NP}$-completitud

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Enero 2014

Bibliografía

- R. Neapolitan y K. Naimipour. *Foundations of Algorithms using C++ pseudocode*. Tercera edición. Jones and Bartlett Publishers, 2004.
  Capítulos 7, 8 y 9
- E. Horowitz, S. Sahni y S. Rajasekaran. *Computer Algorithms*. Tercera edición. Computer Science Press, 1998.
  Capítulos 10, 11 y 12
- G. Brassard y P. Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.
  Capítulo 12

Introducción

- **Algoritmia:** Se encarga del *diseño* y *análisis* de algoritmos específicos (cada vez más eficientes) para resolver un problema dado.
- **Complejidad Computacional:** considera globalmente *todos* los posibles algoritmos para resolver un problema dado.
- Estamos interesados en la distinción que existe entre los problemas que pueden ser resueltos por un algoritmo en tiempo polinómico y los problemas para los cuales no conocemos ningún algoritmo polinómico (es decir, el mejor algoritmo conocido es no-polinómico).
- La Teoría de la $\mathcal{NP}$-completitud no proporciona un método para obtener algoritmos de tiempo polinómico. Ni dice que estos algoritmos no existan. Lo que muestra es que muchos de los problemas para los cuales no conocemos algoritmos polinómicos están relacionados (computacionalmente).

Introducción

- Dos clases de problemas: $\mathcal{NP}$-completos y $\mathcal{NP}$-difíciles.
- Un problema $\mathcal{NP}$-completo tiene la propiedad de que puede ser resuelto en tiempo polinómico si y solo si todos los problemas $\mathcal{NP}$-completos pueden ser resueltos en tiempo polinómico.
- Si un problema $\mathcal{NP}$-difícil puede ser resuelto en tiempo polinómico, entonces todos los problemas $\mathcal{NP}$-completos pueden ser resueltos en tiempo polinómico.

Cotas inferiores

- Algoritmia: Demostrar (expresando y analizando un algoritmo concreto) que el problema bajo estudio puede resolverse en un tiempo que está en $O(f(n))$, para alguna función $f(n)$ (que se intenta reducir al máximo).
- Complejidad Computacional: Hallar una función $g(n)$ lo más grande posible tal que se pueda demostrar que *cualquier* algoritmo que resuelva correctamente el problema necesita un tiempo en $\Omega(g(n))$.
  Se obtiene una cota inferior de la complejidad del problema (no de un algoritmo).
- Cuando $f(n) \in \Theta(g(n))$ estaremos muy satisfechos, pues habremos encontrado un algoritmo lo más eficiente posible (salvo por constantes multiplicativas ocultas).
  Ejemplo: Problema de ordenación.

Cotas inferiores

- Obtener cotas inferiores no es fácil: hablan de *todos* los algoritmos que resuelven el problema.
- Sin embargo, para muchos problemas es fácil ver que existe una cota inferior igual a n, donde n es el número de entradas (o salidas).
- Por ejemplo, consideremos todos los algoritmos que encuentran el máximo en un vector desordenado. Todos los valores tienen que ser examinados al menos una vez, por lo que $T(n) \in \Omega(n)$.
- Cotas de este estilo se llaman triviales.
- **Ejemplo:** Problema de la multiplicación de matrices $n \times n$:
  - Una cota inferior es $\Omega(n^2)$.
  - ¿Es posible encontrar un algoritmo $\Theta(n^2)$?
  - De momento, el mejor algoritmo es $\Theta(n^{2.38})$, por lo que se puede seguir investigado para encontrar un algoritmo mejor o encontrar una cota inferior más alta.

Juego de las 20 preguntas

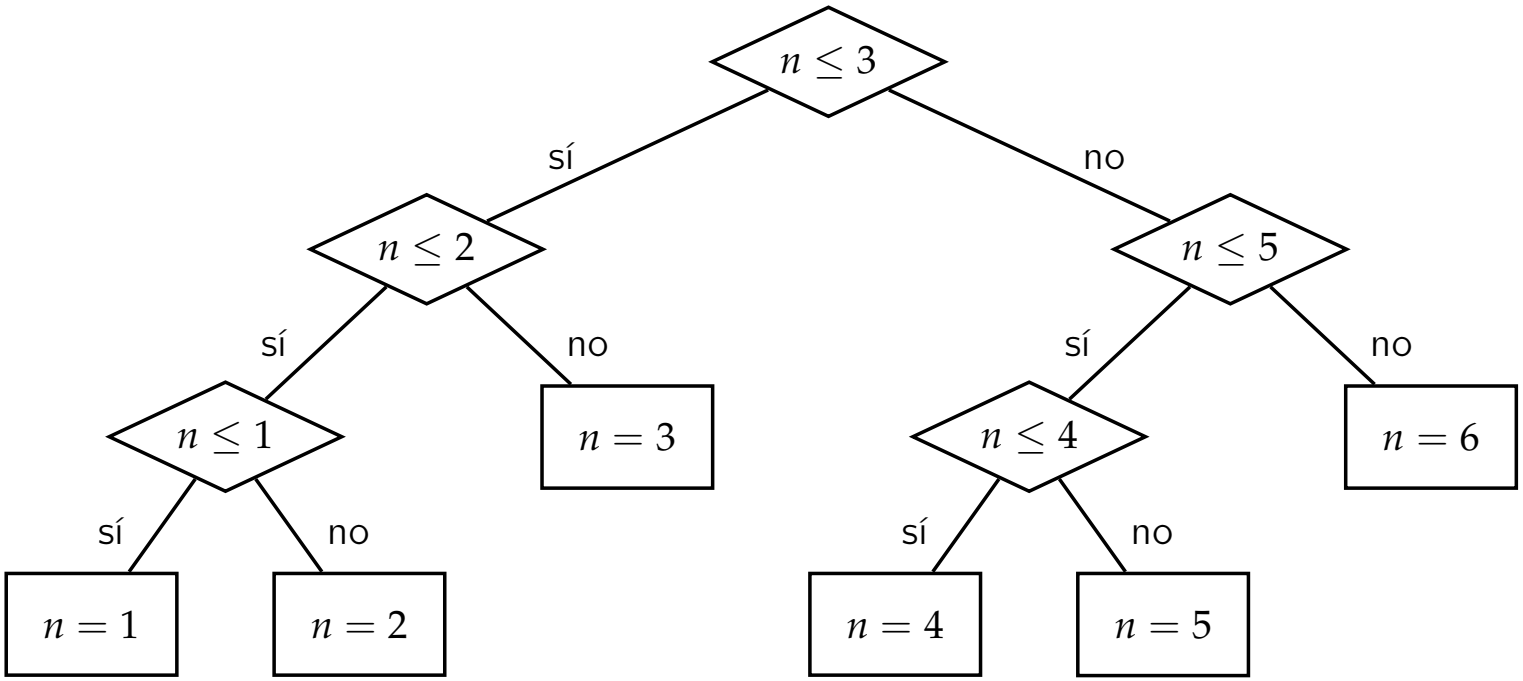
- Seleccionar un número $n \in \{1..1.000.000\}$ y adivinarlo con un máximo de 20 preguntas de respuesta del tipo "sí" / "no".
- ¿Es primo? ¿Es par?
- Algoritmia: encontrar un algoritmo y ver que 20 preguntas de este tipo son suficientes.
  - Divide y vencerás: dividir en dos el conjunto de candidatos.
  - ¿entre 1 y 500.000?
  - $1.000.000 < 2^{20} \Rightarrow$ 20 preguntas son suficientes.
- Complejidad Computacional: demostrar que 20 preguntas son necesarias.
- Varias técnicas:
  - Árboles de decisión.
  - Método del adversario.

Árboles de decisión

- Representan el funcionamiento de un algoritmo concreto para todos los datos posibles de un tamaño dado.
- Utilizamos árboles binarios:
  - Un nodo interno representa una prueba que se aplica a los datos.
  - Una hoja representa un *veredicto* (salida).
  - Una ejecución consiste en un viaje desde la raíz haciendo la pregunta que haya allí. Si la respuesta es "sí" pasamos al hijo izquierdo y si es "no" pasamos al derecho. La ejecución termina al llegar a una hoja, y el veredicto es el resultado.

Juego de las 3 preguntas

- Reducimos a $n \in \{1..6\}$. Árbol concreto (para un algoritmo concreto):



- La altura del árbol corresponde al número de preguntas en el caso peor y tiene que haber una hoja (al menos) para cada posible veredicto.
- Árbol binario de altura $k \Rightarrow$ máximo de 2^k hojas.
- $2^{19} = 524.288 < 10^6 \Rightarrow$ 20 preguntas son necesarias (no quiere decir que sean suficientes!).

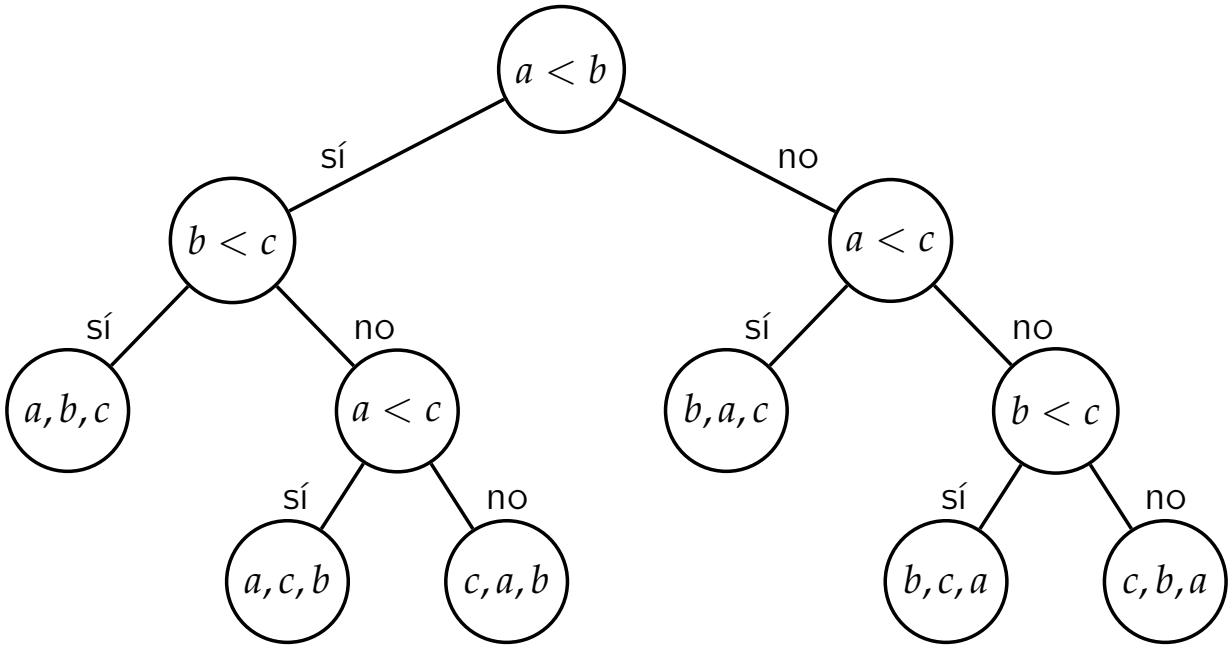
Cota inferior de la ordenación basada en comparaciones

Vamos a ver que el número de comparaciones necesarias para ordenar n valores distintos está en $\Omega(n \log n)$, si para ordenar nos limitamos a utilizar comparaciones entre elementos.

Primero consideremos el siguiente algoritmo que ordena tres valores distintos:

```
proc ordenar-tres(a,b,c)
    si a < b entonces
        si b < c entonces ⟨a,b,c⟩ := ⟨a,b,c⟩
        si no si a < c entonces ⟨a,b,c⟩ := ⟨a,c,b⟩
                           si no ⟨a,b,c⟩ := ⟨c,a,b⟩ fsi
        fsi
    si no { b < a }
        si a < c entonces ⟨a,b,c⟩ := ⟨b,a,c⟩
        si no si b < c entonces ⟨a,b,c⟩ := ⟨b,c,a⟩
                           si no ⟨a,b,c⟩ := ⟨c,b,a⟩ fsi
        fsi
    fsi
fproc
```

Podemos asociar un árbol binario al procedimiento `ordenar-tres` que indique cómo se van haciendo las comparaciones.



En el árbol hay una hoja por cada posible ordenación.

El árbol de decisión se dice que es válido para ordenar n valores si para cada permutación de los n valores hay un camino desde la raíz a una hoja que ordena la permutación. Debe tener al menos $n!$ hojas.

El número de comparaciones en el caso peor hechas por un árbol de decisiones es igual a su profundidad.

Encontrar una cota inferior de la profundidad de un árbol binario que contiene $n!$ hojas.

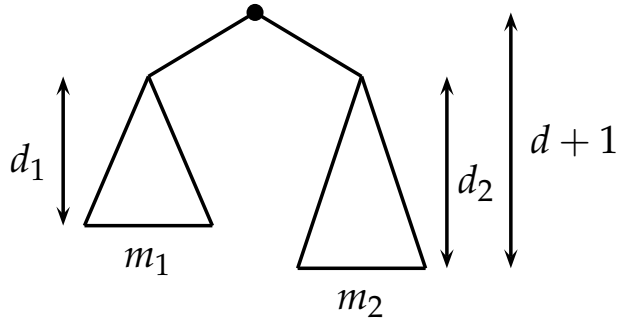
Si m es el número de hojas de un árbol binario y d es su profundidad, entonces se cumple

$$d \geq \lceil \log m \rceil.$$

Demostramos primero que para un árbol binario de decisión con m hojas y profundidad d se cumple $m \leq 2^d$, por inducción sobre d.

Base: Cuando $d = 0$, solo existe un nodo que es una hoja, y $1 \leq 2^0$.

Paso inductivo: Suponemos cierto que para todo árbol binario con m' hojas y profundidad $d' \leq d$ se cumple $m' \leq 2^{d'}$. Probamos para profundidad $d+1$ que $m \leq 2^{d+1}$, donde m es el número de hojas.



donde $d = \max\{d_1, d_2\}$. Por hipótesis de inducción,

$$m_1 \leq 2^{d_1} \quad \wedge \quad m_2 \leq 2^{d_2}.$$

Por lo que $m = m_1 + m_2 \leq 2^{d_1} + 2^{d_2} \leq 2 \cdot 2^{\max\{d_1,d_2\}} = 2^{d+1}$.

Tomando ahora logaritmos, obtenemos $d \geq \log m$ y como d es entero, esto implica $d \geq \lceil \log m \rceil$.

De todo lo anterior se deduce que cualquier algoritmo que ordene n valores distintos utilizando exclusivamente comparaciones, debe realizar en el caso peor al menos $\lceil \log(n!) \rceil$ comparaciones.

¿Cómo de grande es $\log(n!)$?

$$\begin{aligned}
\log(n!) &= \log[n(n-1)(n-2)\cdots(2)1] \\
&= \sum_{i=2}^{n} \log i \\
&\geq \int_1^n \log x \mathrm{d}x = \frac{1}{\ln 2}\left[x \ln x - x\right]_{x=1}^{x=n} \\
&= \frac{1}{\ln 2}(n \ln n - n + 1) \\
&\geq n \log n - 1.45n.
\end{aligned}$$

Método del adversario

- Especie de juego en el que el algoritmo se enfrenta a un diablillo malévolo que intenta hacerle trabajar lo máximo posible.
- Se pone en marcha el algoritmo con una entrada inicialmente no específica (salvo su tamaño).
- A medida que el algoritmo va examinando la entrada, el diablillo (adversario) responde de forma que haga trabajar al algoritmo lo máximo posible, pero siendo siempre consecuente.
- El diablillo mantiene la incertidumbre sobre la respuesta correcta el mayor tiempo posible.
- Si el algoritmo afirma conocer la respuesta antes de haber examinado la entrada suficientemente, el diablillo podrá mostrar una entrada válida, consistente con sus respuestas, pero con solución correcta diferente a la salida del algoritmo.
- El algoritmo puede equivocarse si no examina suficientemente la entrada en el caso peor.

Juego de las 20 preguntas

- Seleccionar un número entre 1 y 1.000.000, pero si el adversario hace "trampas", puede retrasar su elección mientras sea posible, haciendo que dependa de las preguntas realizadas, y siempre respondiendo consistentemente.
- $Q_i \equiv$ i-ésima pregunta (puede depender de las respuestas anteriores).
  $Q_i(n) \equiv$ respuesta siendo n el número buscado.
  $S_i \equiv$ conjunto de candidatos tras Q_i. Y $k_i = |S_i|$.
  $S_0 = \{1..1.000.000\}$, $k_0 = 10^6$.
  $Y_i = \{n \in S_{i-1} \mid Q_i(n) = \text{“sí”}\}$.
  $N_i = \{n \in S_{i-1} \mid Q_i(n) = \text{“no”}\}$.
  $N_i \cap Y_i = \emptyset$ $N_i \cup Y_i = S_{i-1}$ $|S_{i-1}| = |N_i| + |Y_i|$
  Al menos Y_i o N_i contiene $\lceil k_{i-1}/2 \rceil$ números o más, y por lo tanto, puede ocurrir que $k_i \geq \lceil k_{i-1}/2 \rceil$ $\forall i$.

Juego de las 20 preguntas

- El adversario responde "sí" solo si $|Y_i| > |N_i|$. En tal caso,

$$S_i := Y_i.$$

- En caso contrario,

$$S_i := N_i.$$

- Decidirá el número solo cuando quede un único candidato.
- Con cada pregunta el número de candidatos se divide como mucho por 2, y con menos de 20 preguntas siempre queda más de un candidato.
- Si intentamos adivinar antes, siempre nos podrían decir que hemos fallado.

Búsqueda del máximo de un vector

- Generalmente los árboles de decisión son más fáciles de utilizar que el método del adversario, pero a veces no resultan útiles.

```
fun índice-máx(V[1..n]) dev k
    m := V[1]      { máximo por ahora }
    k := 1         { índice del máximo }
    para i = 2 hasta n hacer
        si V[i] > m entonces
            m := V[i] ; k := i
        fsi
    fpara
ffun
```

- $n-1$ comparaciones son suficientes. ¿Se puede hacer mejor?

Búsqueda del máximo de un vector: Árbol de decisión

- Árbol de decisión:
  - Debe contener n veredictos posibles, y como es un árbol binario, tendrá una altura mínima $\lceil \log n \rceil$.
  - Un algoritmo basado en comparaciones tiene como cota inferior $\lceil \log n \rceil$ comparaciones (en el caso peor).
    Muy lejos de $n - 1$.
- ¿Existirá algún algoritmo que realice un número de comparaciones en $\Theta(\log n)$?

Búsqueda del máximo de un vector: Método del adversario

- Método del adversario:
  - Algoritmo arbitrario, aplicado a $V[1..n]$, y el diablillo responde a todas las preguntas de comparación entre elementos como si $V[i] = i$.
  - Si preguntamos ¿$V[i] < V[j]$? con $i \neq j$, $\mathbf{mín}(i,j)$ "pierde" en esta comparación.
  - Si el algoritmo efectúa menos de $n-1$ comparaciones antes de responder k, tomamos $j \neq k$, $1 \leq j \leq n$, que no haya perdido ninguna comparación.
  - Tal j existe porque como máximo se han hecho $n-2$ comparaciones y cada una crea como máximo un perdedor.
  - Así que el diablillo podrá afirmar que j es la respuesta y no k (el algoritmo ha fallado).
    El diablillo muestra la entrada: $V[i] = i \quad \forall\, i \neq j, \qquad V[j] = n+1$
    Y por tanto, $V[k] = k < V[j]$.
  - Conclusión: $n-1$ comparaciones son necesarias.

Intratabilidad

Definición: Un algoritmo de tiempo polinómico es uno cuya complejidad en tiempo en el caso peor está acotada superiormente por un polinomio (en el tamaño de la entrada).

- Son polinómicos los de complejidad: $2n, \quad n + n^5, \quad n \log n$.
- No son polinómicos los de complejidad: $2^n, \quad 2^{\sqrt{n}}, \quad n!$.
- Un problema es intratable si no se puede resolver en tiempo polinómico. (Es una propiedad del problema.)
- Tres categorías generales de problemas en lo que se refiere a la intratabilidad:
  - Problemas para los que se conoce un algoritmo polinómico.
  - Problemas que se han demostrado intratables.
  - Problemas que no se ha demostrado que sean intratables, pero para los cuales no se ha encontrado un algoritmo polinómico.

Clase 1: Polinómicos

- Cualquier problema para el que hayamos encontrado un algoritmo polinómico.
  - Ordenación: $\Theta(n \log n)$.
  - Búsqueda en un vector ordenado: $\Theta(\log n)$.
  - Multiplicación encadenada de matrices: $\Theta(n^3)$.
  - Problema de los caminos más cortos: $\Theta(n^3)$.
- Para muchos de estos problemas hay algoritmos no polinómicos que los resuelven. Pero eso no quiere decir que los problemas no sean polinómicos.

Clase 2: Intratables

- Problemas para los que se ha demostrado su intratabilidad. Dos tipos:
- Problemas que requieren una cantidad no polinómica de salidas.
  - Determinar todos los ciclos Hamiltonianos de un grafo con n vértices. Podría haber $(n-1)!$ ciclos.
- Problemas que no requieren salida no polinómica pero podemos probar que no se pueden resolver en tiempo polinómico. Por extraño que parezca, se han encontrado pocos problemas de este tipo.
  - Problemas indecidibles: no existe algoritmo que los resuelva. Problema de parada.
  - Problemas decidibles: generalmente construidos "artificialmente".

Clase 3: Ni intratables, ni polinómicos ?

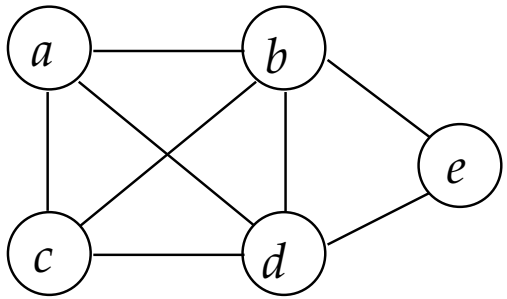
- Problemas para los cuales no se ha encontrado un algoritmo polinómico pero nadie ha demostrado que dicho algoritmo no sea posible.
  - Problema de la mochila.
  - Problema del viajante.
  - Problema de la suma de subconjuntos.
  - Problema del coloreado de grafos.
  - Problema de los ciclos Hamiltonianos.
  - Problema de los envases.
- Hay una relación interesante entre muchos de estos problemas: la estudia la teoría de la $\mathcal{NP}$-completitud.

Teoría de la $\mathcal{NP}$-completitud

- Inicialmente nos restringimos a problemas de decisión: la salida de un problema de decisión es simplemente "sí" o "no".
- Muchos de los problemas anteriores se presentaron como problemas de optimización. Cada problema de optimización tiene un problema de decisión correspondiente.
- EJEMPLOS:
  - Problema del Viajante: Dado un grafo dirigido y valorado, el problema de optimización del viajante consiste en encontrar un circuito de coste mínimo que empiece en un vértice, acabe en ese vértice, y visite al resto de vértices exactamente una vez.
    El problema de decisión del viajante consiste en determinar, dado un positivo d, si el grafo tiene un circuito de coste no mayor que d.
    El problema de decisión correspondiente no es más difícil que el de optimización.

Teoría de la $\mathcal{NP}$-completitud

- Problema del coloreado de grafos: Determinar el mínimo número de colores necesarios para colorear un grafo de tal forma que no haya dos vértices adyacentes con el mismo color.
  El problema de decisión correspondiente consiste en, dado un entero m, determinar si existe un coloreado que utilice como mucho m colores.
- Problema del Cliqué: Un cliqué de un grafo no dirigido $G = \langle V, A \rangle$ es un subconjunto W de V tal que cada vértice de W es adyacente al resto de vértices en W.



El problema de optimización del cliqué consiste en encontrar el tamaño de un cliqué maximal.
El problema de decisión del cliqué consiste en determinar, dado un entero positivo k, si existe un cliqué que contenga al menos k vértices.

Teoría de la $\mathcal{NP}$-completitud

- No se han encontrado algoritmos polinómicos ni para el problema de decisión ni para el de optimizacion de los ejemplos anteriores.
- Sin embargo, si tuvieramos un algoritmo polinómico para el problema de optimización tendríamos también un algoritmo polinómico para el correspondiente problema de decisión.
- Una solución al problema de optimización produce una solución del correspondiente problema de decisión.
- Por eso inicialmente podemos restringirnos a problemas de decisión, para pasar después a los problemas de optimización.

Las clases $\mathcal{P}$ y $\mathcal{NP}$

Definición: $\mathcal{P}$ es el conjunto de todos los problemas de decisión que pueden ser resueltos por un algoritmo polinómico.

- ¿Qué problemas de decisión están en $\mathcal{P}$?
- ¿Qué problemas de decisión no están en $\mathcal{P}$?

- Un algoritmo no determinista está compuesto por dos fases:
  1. Fase de adivinación (no determinista): Dada una instancia del problema, produce una salida S que puede entenderse como una supuesta solución.
  2. Fase de verificación (determinista): Dada la instancia y la salida S, y procediendo de forma determinista, o devuelve `cierto`, lo cual significa que se ha verificado que la respuesta para esta instancia es "sí", o devuelve `falso`.

Las clases $\mathcal{P}$ y $\mathcal{NP}$

- Aunque nunca podamos utilizar un algoritmo no determinista para resolver un problema, decimos que un algoritmo no determinista "resuelve" un problema de decisión si:
  1. Para cualquier instancia para la cual la respuesta es "sí" hay alguna salida S para la cual la fase de verificación devuelve `cierto`.
  2. Para cualquier instancia para la cual la respuesta es "no" no hay ninguna salida S para la cual la fase de verificación devuelva `cierto`.

Problema de decisión del viajante

- Supongamos que para un grafo G y un valor d, una persona asegura que la respuesta al problema es "sí", es decir, asegura que hay un circuito con coste total no superior a d.
- Es razonable que le pidamos que "pruebe" su afirmación produciendo un circuito con coste no superior a d.
- Si la persona produce un presunto ciclo, podríamos construir un algoritmo que verificara si la salida es un circuito con coste no superior a d.

```
fun verificar(G,d,S) dev r : bool
    si S es un circuito ∧ coste-total(S) ≤ d entonces
        r := cierto
    si no
        r := falso
    fsi
ffun
```

Ejercicio: Implementar el algoritmo con más detalle y mostrar que es polinómico.

La Clase $\mathcal{NP}$

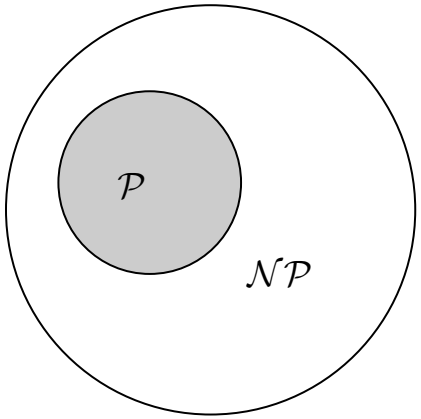
Definición: Un algoritmo no determinista de tiempo polinómico es un algoritmo no determinista cuya fase de verificación es un algoritmo de tiempo polinómico.

Definición: $\mathcal{NP}$ es el conjunto de todos los problemas de decisión que pueden ser resueltos por algoritmos no deterministas de tiempo polinómico.

- $\mathcal{NP}$ significa "nondeterministic polynomial".
- Para que un problema esté en $\mathcal{NP}$ debe existir un algoritmo que haga la verificación en tiempo polinómico.
- Ya que `verificar` lo hace, el problema de decisión del viajante está en $\mathcal{NP}$.

La Clase $\mathcal{NP}$

- Esto no significa que haya algoritmos de tiempo polinómico que lo resuelvan.
- El propósito de introducir los conceptos de algoritmo no determinista y $\mathcal{NP}$ es el de clasificar los algoritmos.
- Hay mejores formas de resolver un algoritmo que ir generando soluciones y verificarlas.
- ¿Qué otros problemas de decisión están en $\mathcal{NP}$? Los vistos en los ejemplos anteriores. (Y muchísimos más.)
- Hay un gran número de problemas que están trivialmente en $\mathcal{NP}$: todo problema en $\mathcal{P}$ está también en $\mathcal{NP}$. ¿Por qué?
- La figura muestra la relación que se cree existe entre las clases $\mathcal{P}$ y $\mathcal{NP}$.



- Aunque nadie ha probado que exista un problema en $\mathcal{NP}$ que no esté en $\mathcal{P}$.

La Clase $\mathcal{NP}$

- La cuestión de si $\mathcal{P} = \mathcal{NP}$ es una de las más misteriosas e importantes en informática teórica.
- Para mostrar que $\mathcal{P} \neq \mathcal{NP}$ tendríamos que encontrar un problema en $\mathcal{NP}$ que no esté en $\mathcal{P}$, mientras que para probar que $\mathcal{P} = \mathcal{NP}$, en principio, tendríamos que encontrar un algoritmo polinómico para cada problema en $\mathcal{NP}$.
- Veremos que esta tarea se puede simplificar considerablemente. Veremos que es suficiente encontrar un algoritmo de tiempo polinómico para uno solo de los problemas en la clase de los problemas $\mathcal{NP}$-completos.

Problema SAT-FNC

- Una variable lógica es una variable que puede tomar los valores `cierto` o `falso`. Si x es una variable lógica, $\overline{x}$ es la negación de x.
- Un literal es una variable lógica o su negación.
- Una cláusula es una secuencia de literales separados por el operador lógico `or` ($\vee$).
- Una expresión lógica en forma nomal conjuntiva (FNC) es una secuencia de cláusulas separadas por el operador lógico `and` ($\wedge$).
- El problema de decisión de satisfactibilidad FNC (*SAT-FNC*) consiste en determinar, dada una expresión lógica en FNC, si existe alguna asignación de valores `cierto` y `falso` a las variables, que haga la expresión cierta.

Problema SAT-FNC

Ejemplos:

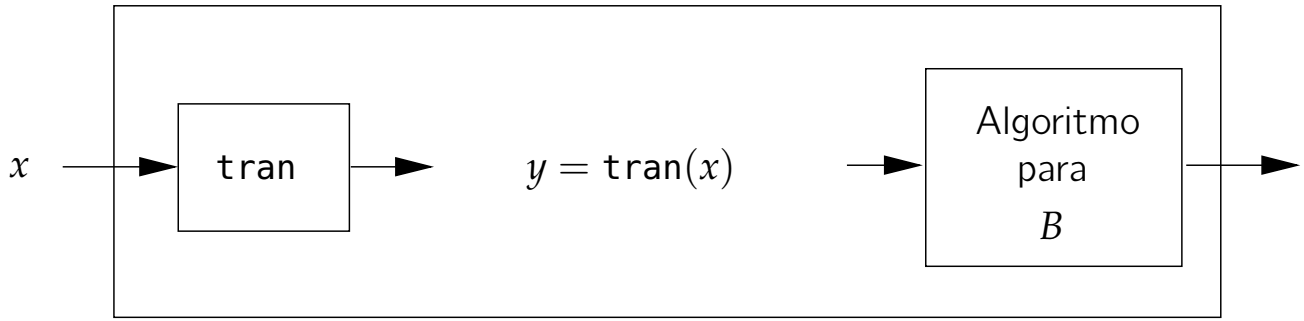
- Para la instancia
$$(x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge \overline{x_2}$$
la respuesta es "sí", ya que la asignación $x_1 =$ `cierto`, $x_2 =$ `falso` y $x_3 =$ `falso` hace la expresión cierta.
- Para la instancia
$$(x_1 \vee x_2) \wedge \overline{x_1} \wedge \overline{x_2}$$
la respuesta es "no".

Problema SAT-FNC

- Es fácil escribir un algoritmo *polinómico* que tome como entrada una expresión lógica en FNC y una asignación a las variables y verifique si la expresión es cierta para esa asignación.
  Ejercicio: Escribirlo.
- Por tanto, *SAT-FNC* está en $\mathcal{NP}$.
- Nadie ha encontrado un algoritmo polinómico para *SAT-FNC* y nadie ha probado que dicho algoritmo no exista, por lo que no sabemos si está en $\mathcal{P}$.
- En 1971, Stephen Cook demostró que si *SAT-FNC* está en $\mathcal{P}$, entonces $\mathcal{P} = \mathcal{NP}$.
- Esta es una característica de los problemas $\mathcal{NP}$-completos (hay muchos más).

Reducibilidad polinómica

- Supongamos que queremos resolver el problema de decisión A y que tenemos un algoritmo que resuelve el problema de decisión B.
- Supongamos también que tenemos un algoritmo que construye una instancia y de B para toda instancia x de A de tal forma que un algoritmo para B responde "sí" para y si y solo si la respuesta al problema A para x es "sí". Dicho algoritmo se denomina algoritmo de transformación.
- El algoritmo de transformación combinado con el algoritmo para B nos da un algoritmo para A.



Algoritmo para A

Reducibilidad polinómica

Definición: Si existe un algoritmo de transformación polinómico del problema de decisión A en el problema de decisión B, el problema A es reducible polinómicamente al problema B. Lo denotamos

$$A \leq^p B.$$

- Si el algoritmo de transformación es polinómico y tenemos un algoritmo polinómico para B, intuitivamente parece que el algoritmo para A resultante de la combinación debe ser polinómico.

Teorema 1

Si el problema de decisión B está en $\mathcal{P}$ y $A \leq^p B$, entonces el problema de decisión A está en $\mathcal{P}$.

- *Demostración*:
  - Sea p el polinomio que acota la complejidad en tiempo del algoritmo de transformación, y q el polinomio que acota la complejidad del algoritmo polinómico para B.
  - Supongamos que tenemos una instancia para A de tamaño n.
  - Como el algoritmo de transformación da como mucho $p(n)$ pasos, el tamaño de la instancia del problema B es como mucho $p(n)$.
  - El algoritmo para B realiza como mucho $q(p(n))$ pasos.
  - Por tanto, la cantidad total de trabajo para resolver A es como mucho $p(n) + q(p(n))$, que es un polinomio en n. □

Problemas $\mathcal{NP}$-completos

Definición: Un problema B es $\mathcal{NP}$-completo si

1. está en $\mathcal{NP}$, y
2. para cualquier otro problema A en $\mathcal{NP}$, $A \leq^p B$.

- Por el Teorema 1, si pudieramos mostrar que un problema $\mathcal{NP}$-completo cualquiera está en $\mathcal{P}$, podríamos concluir que $\mathcal{P} = \mathcal{NP}$.

Teorema 2 (Teorema de Cook)

SAT-FNC es $\mathcal{NP}$-completo. Demostrado en [HSR98, pág. 508].

- Aunque no veamos la demostración, esta no consiste en reducir individualmente todo problema de $\mathcal{NP}$ a *SAT-FNC*. La demostración se basa en propiedades comunes a todos los problemas en $\mathcal{NP}$, para probar que cualquier problema en $\mathcal{NP}$ debe poder ser reducible a *SAT-FNC*.
- Una vez probado el Teorema 2, se puede probar que muchos otros problemas son $\mathcal{NP}$-completos. Las demostraciones se basan en el siguiente teorema.

Teorema 3

Un problema C es $\mathcal{NP}$-completo si

1. está en $\mathcal{NP}$, y
2. para algún problema $\mathcal{NP}$-completo B, $B \leq^p C$.

Demostración:

- Por ser B $\mathcal{NP}$-completo, para cualquier problema A en $\mathcal{NP}$, $A \leq^p B$.
- La reducibilidad es transitiva. Por tanto, $A \leq^p C$.
- Ya que C está en $\mathcal{NP}$, concluimos que C es $\mathcal{NP}$-completo. □

- Por los Teoremas 2 y 3, podemos demostrar que un problema es $\mathcal{NP}$-completo probando que está en $\mathcal{NP}$ y que *SAT-FNC* se reduce a él.
- *SAT-FNC* es la *semilla* para ir encontrando problemas $\mathcal{NP}$-completos.

PDC es $\mathcal{NP}$-completo

- *PDC* $\equiv$ Problema de Decisión del Cliqué.

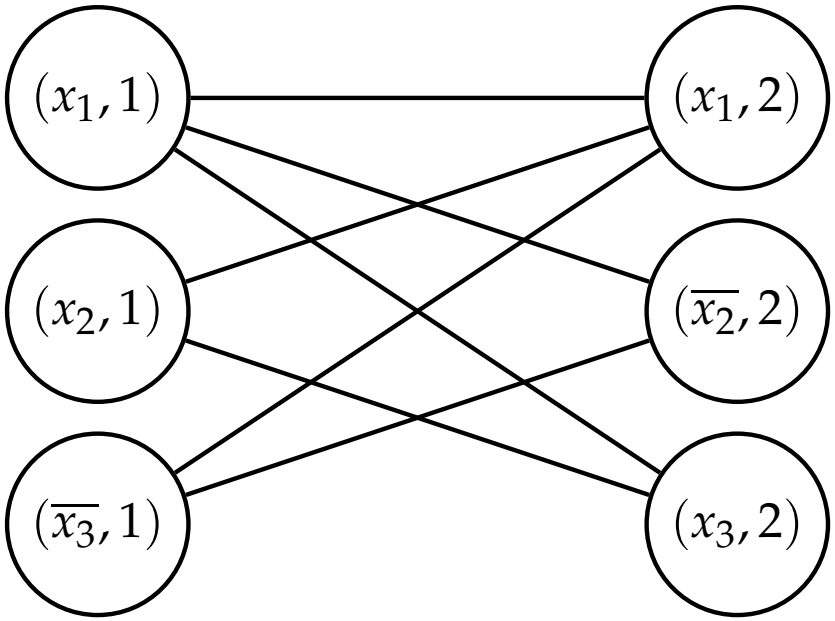
  Ejercicio: Probar que *PDC* está en $\mathcal{NP}$ escribiendo un algoritmo de verificación polinómico.
- Tenemos que probar que *SAT-FNC* $\leq^p$ *PDC*.
- Sea $B = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ una expresión lógica en FNC, con variables $x_1, x_2, \ldots, x_n$.
- Transformamos B en el grafo $G = \langle V, A \rangle$ con

$$V = \{(y,i) \mid y \text{ es un literal en la cláusula } C_i\},$$

$$A = \{\langle (y,i),(z,j) \rangle \mid i \neq j \text{ y } \overline{z} \neq y\}.$$

PDC es $\mathcal{NP}$-completo

- Por ejemplo, para $B = \underbrace{(x_1 \vee x_2 \vee \overline{x_3})}_{C_1} \wedge \underbrace{(x_1 \vee \overline{x_2} \vee x_3)}_{C_2}$ el grafo G es:



Ejercicio: Probar que la transformación es polinómica.

PDC es $\mathcal{NP}$-completo

- Supongamos la transformación de instancias: $B \mapsto \langle G, k \rangle$.
- Tenemos que probar que B es satisfactible si y solo si G tiene un cliqué de tamaño al menos k.

$\Rightarrow$ Si B es satisfactible, entonces G tiene un cliqué de tamaño al menos k.

- Si B es satisfactible, hay una asignación a las variables que hace cada cláusula cierta.
- Esto significa que en dicha asignación hay al menos un literal en cada cláusula C_i que es `cierto`. Elegimos uno de estos literales de cada C_i.
- Sea $V' = \{(y, i) \mid y$ es el literal `cierto` tomado de $C_i\}$.
- V' forma un cliqué en G de tamaño k, ya que hay una arista entre cada par de vértices (y, i) y (z, j) en V', porque $i \neq j$ y tanto y como z son ciertos.

PDC es $\mathcal{NP}$-completo

$\Leftarrow$ Si G tiene un cliqué de tamaño al menos k, entonces B es satisfactible.

- Ya que no hay arista entre un vértice (y,i) y otro (z,i), los índices de los vértices en el cliqué tienen que ser todos diferentes.
- Ya que solo hay k índices diferentes, el cliqué tiene que tener como mucho k vértices.
- Por tanto, si el grafo G tiene un cliqué V' de tamaño al menos k, el número de vértices en V' tiene que ser exactamente k.
- Tomamos $S = \{y \mid (y,i) \in V'\}$.
- S contiene k literales, uno por cada cláusula.
- S no puede contener un literal y y su complementario $\overline{y}$ porque no hay ninguna arista conectando (y,i) con $(\overline{y},j)$ para ningún i y j.
- Por tanto, si hacemos

$$x_i = \begin{cases} \texttt{cierto} & \text{si } x_i \in S \\ \texttt{falso} & \text{si } \overline{x_i} \in S \end{cases}$$

  y asignamos valores arbitrarios a las variables que no están en S, todas las cláusulas de B son ciertas.
- Por tanto, B es satisfactible.

PDCH es $\mathcal{NP}$-completo

- *PDCH* $\equiv$ Problema de Decisión de los Ciclos Hamiltonianos: Determinar si un grafo tiene un ciclo que pase por todos los vértices exactamente una vez y vuelva al vértice inicial.
- *PDCH* está en $\mathcal{NP}$.
- *SAT-FNC* $\leq^p$ *PDCH*, demostrado en [HSR98, pág. 522].
- Por tanto, *PDCH* es $\mathcal{NP}$-completo.

PDV es $\mathcal{NP}$-completo

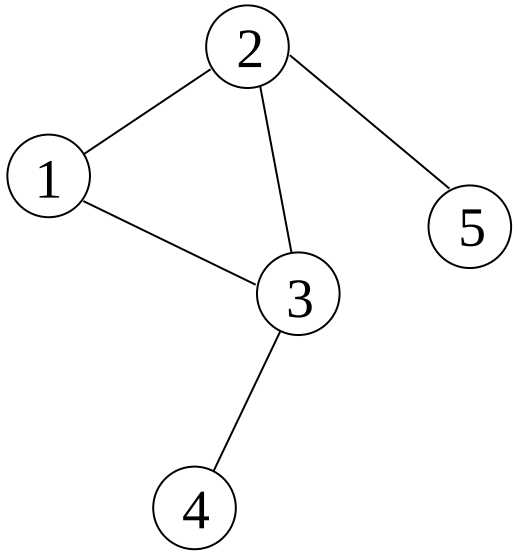
- *PDV* $\equiv$ Problema de Decisión del Viajante (con grafos no dirigidos).
- La función `verificar` ya vista sirve también para grafos no dirigidos. Por tanto, *PDV* está en $\mathcal{NP}$.
- Solo nos queda por probar que algún problema $\mathcal{NP}$-completo se reduce a *PDV*.
- Probamos que *PDCH* $\leq^p$ *PDV*.
- Transformamos una instancia $G = \langle V, A \rangle$ de *PDCH* en una instancia $G' = \langle V, A' \rangle$, con el mismo conjunto de vértices V, una arista entre cada par de vértices, y el siguiente coste:

$$coste(\langle u, v \rangle) = \begin{cases} 1 & \text{si } \langle u, v \rangle \in A \\ 2 & \text{si } \langle u, v \rangle \notin A \end{cases}$$

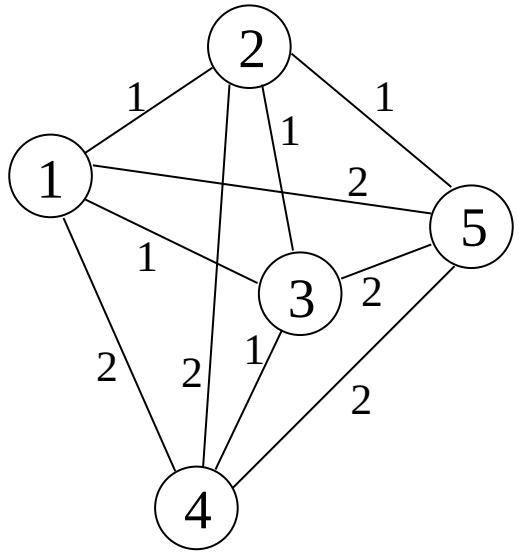
Exactamente, $G \mapsto \langle G', n \rangle$

PDV es $\mathcal{NP}$-completo

- Por ejemplo,



G

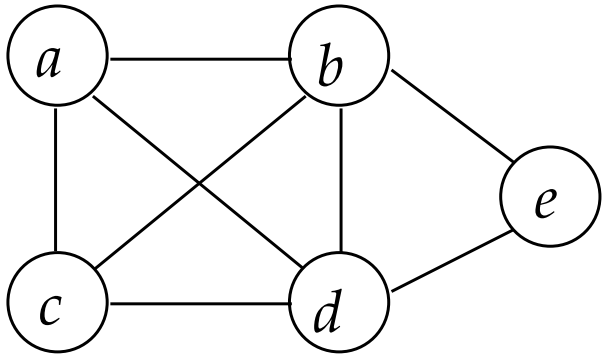


G'

- G tiene un circuito Hamiltoniano si y solo si G' tiene un circuito de longitud no mayor que n, con $n = |V|$.

Problema de la cobertura de vértices

- Una cobertura de un grafo no dirigido $G = \langle V, A \rangle$ es un subconjunto $V' \subseteq V$ tal que si $\langle u, v \rangle \in A$, entonces $u \in V'$ o $v \in V'$ (o ambos).
- Cada vértice "cubre" sus aristas incidentes, y una cobertura es un conjunto de vértices que cubre todas las aristas.
- El tamaño de una cobertura es el número de vértices.



- El Problema de la Cobertura de vértices consiste en encontrar una cobertura de tamaño mínimo.
- El Problema de Decisión de la Cobertura de Vértices consiste en determinar si un grafo tiene una cobertura de tamaño como mucho k.

PDCV es $\mathcal{NP}$-completo

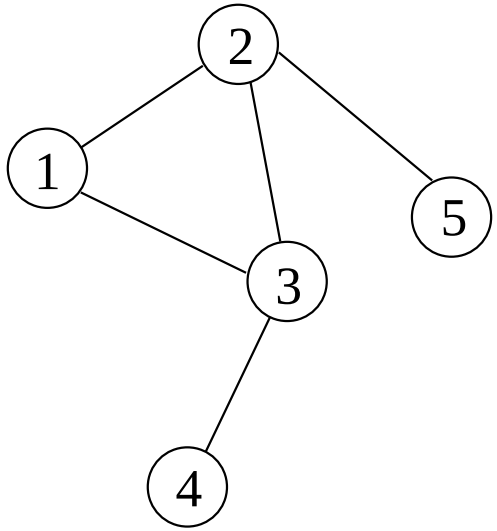
- *PDCV* $\equiv$ Problema de Decisión de la Cobertura de Vértices.

  Ejercicio: Probar que *PDCV* está en $\mathcal{NP}$.
- *PDC* $\equiv$ Problema de Decisión del Cliqué.
  Probamos que *PDC* $\leq^p$ *PDCV*, para concluir que *PDCV* es $\mathcal{NP}$-completo.
- Sean $G = \langle V, A \rangle$ y k una instancia de *PDC*, con $|V| = n$.
- Construimos $\overline{G} = \langle V, \overline{A} \rangle$ donde

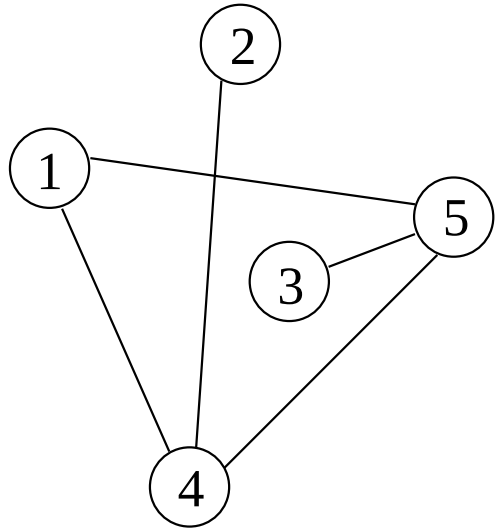
$$\overline{A} = \{\langle u, v \rangle \mid u, v \in V,\ u \neq v \text{ y } \langle u, v \rangle \notin A\}.$$

PDCV es $\mathcal{NP}$-completo

- Por ejemplo,



G



$\overline{G}$

Ejercicio: Probar que la transformación es polinómica.

PDCV es $\mathcal{NP}$-completo

- Transformación: $\langle G, k \rangle \mapsto \langle \overline{G}, n-k \rangle$. Tenemos que probar que G tiene un cliqué de tamaño al menos k si y solo si $\overline{G}$ tiene una cobertura de tamaño como mucho $n-k$.

$\Rightarrow$ Si G tiene un cliqué de tamaño al menos k, entonces $\overline{G}$ tiene una cobertura de tamaño como mucho $n-k$.

- Sea $V' \subseteq V$ un cliqué de G, con $|V'| \geq k$.
- Veamos que $V - V'$ es una cobertura de $\overline{G}$.
- Sea $\langle u, v \rangle$ una arista cualquiera de $\overline{A}$.
- Entonces, $\langle u, v \rangle \notin A$, lo que implica que al menos u o v no pertenecen a V', ya que todo par de vértices de V' está conectado por una arista de A.
- Por tanto, u o v está en $V - V'$, lo que significa que la arista $\langle u, v \rangle$ está cubierta por $V - V'$.
- Concluimos que $V - V'$, de tamaño $\leq n-k$, forma una cobertura de $\overline{G}$.

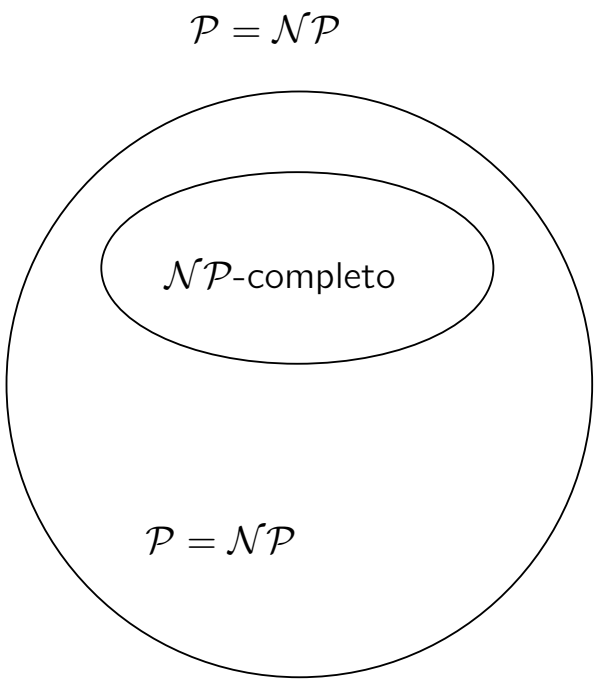
PDCV es $\mathcal{NP}$-completo

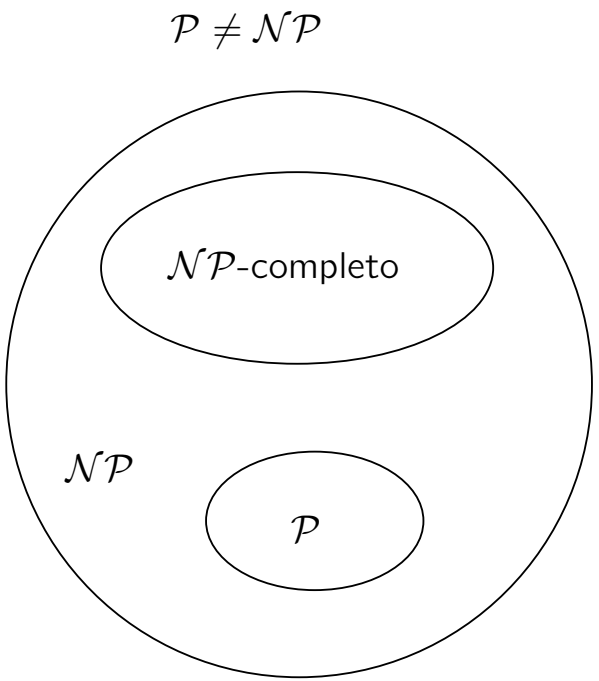
$\Leftarrow$ Si $\overline{G}$ tiene una cobertura de tamaño como mucho $n-k$, entonces G tiene un cliqué de tamaño al menos k.

- Supongamos que $\overline{G}$ tiene una cobertura $V' \subseteq V$, con $|V'| \leq n-k$.
- Entonces, para todo $u,v \in V$, si $\langle u,v \rangle \in \overline{A}$, entonces $u \in V'$ o $v \in V'$ o ambos.
- El contrarrecíproco de esta implicación dice que para todo $u,v \in V$, si $u \notin V'$ y $v \notin V'$, entonces $\langle u,v \rangle \in A$.
- Por tanto, $V - V'$ es un cliqué, y su tamaño es $n - |V'| \geq k$.

El estado de $\mathcal{NP}$

- Todos los problemas $\mathcal{NP}$-completos son $\mathcal{NP}$.
- Pero $\mathcal{NP}$-completo $\neq \mathcal{NP}$. El problema trivial que devuelve siempre cierto está en $\mathcal{NP}$ pero no es $\mathcal{NP}$-completo.





Turing reducción

- Ahora extendemos los resultados a problemas en general.

  Definición: Si el problema A puede resolverse en tiempo polinómico utilizando un algoritmo polinómico hipotético para el problema B, entonces A es Turing reducible en tiempo polinómico a B. Lo denotamos así

  $$A \leq^T B.$$

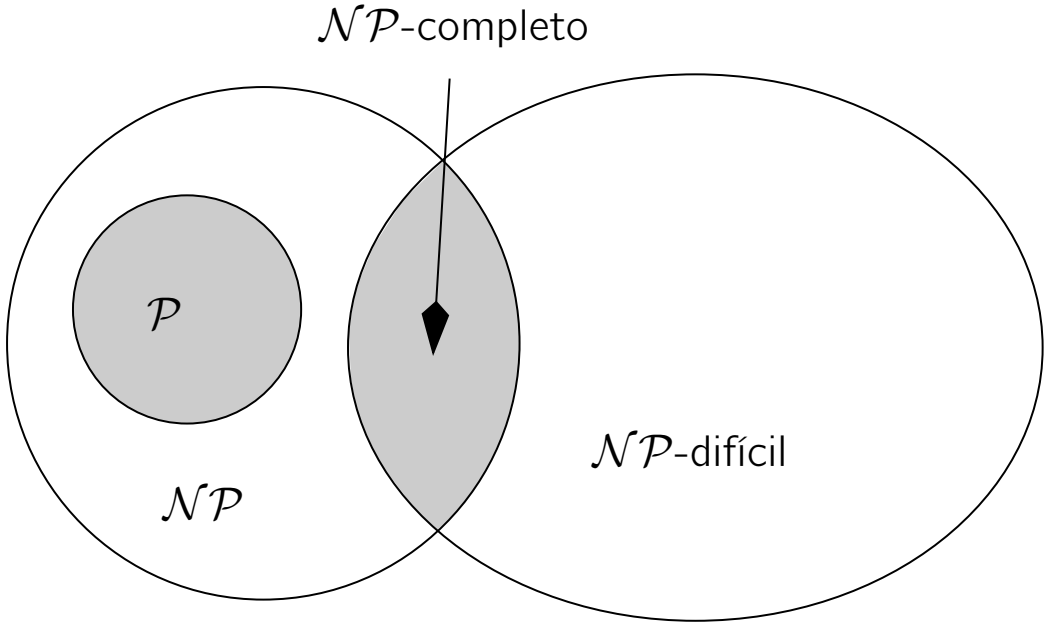
- Claramente, si A y B son problemas de decisión,

  $$A \leq^p B \Longrightarrow A \leq^T B.$$

Problemas $\mathcal{NP}$-difíciles

Definición: Un problema B es $\mathcal{NP}$-difícil si, para algún problema $\mathcal{NP}$-completo A, se cumple $A \leq^T B$.

- Las Turing reducciones son transitivas. Por tanto, todos los problemas en $\mathcal{NP}$ se reducen a algún problema $\mathcal{NP}$-difícil. Esto significa que si existe un algoritmo polinómico para un problema $\mathcal{NP}$-difícil, entonces $\mathcal{P} = \mathcal{NP}$.



Problemas $\mathcal{NP}$-difíciles

- ¿Qué problemas son $\mathcal{NP}$-difíciles?
  - Todo problema $\mathcal{NP}$-completo es $\mathcal{NP}$-difícil. ¿Por qué?
  - Los problemas de optimización correspondientes a problemas $\mathcal{NP}$-completos son $\mathcal{NP}$-difíciles.
- ¿Qué problemas no son $\mathcal{NP}$-difíciles?
  - No sabemos si hay algún problema así.
- De hecho, si demostraramos que algún problema no es $\mathcal{NP}$-difícil, estaríamos probando que $\mathcal{P} \neq \mathcal{NP}$.
  - La razón es que si $\mathcal{P} = \mathcal{NP}$, todo problema en $\mathcal{NP}$ se podría resolver con un algoritmo polinómico.
  - Por tanto, cualquier problema A que esté en $\mathcal{NP}$ podría Turing reducirse en tiempo polínomico a un problema B cualquiera (sin utilizar el algoritmo hipotético para B).
  - Por tanto, todos los problemas serían $\mathcal{NP}$-difíciles.

Problemas $\mathcal{NP}$-difíciles

- Por otro lado, cualquier problema para el cual conocemos un algoritmo polinómico podría no ser $\mathcal{NP}$-difícil.
- De hecho, si probaramos que algún problema para el cual tenemos un algoritmo polinómico fuera $\mathcal{NP}$-difícil, estaríamos probando que $\mathcal{P} = \mathcal{NP}$.
  - La razón es que tendríamos un algoritmo polinómico real en vez de uno hipotético para algún problema $\mathcal{NP}$-difícil.
  - Por tanto, podríamos resolver cada problema en $\mathcal{NP}$ en tiempo polinómico utilizando la Turing reducción del problema al problema $\mathcal{NP}$-difícil.

Tratar problemas $\mathcal{NP}$-difíciles

- ¿Qué podemos hacer con estos problemas? Tres enfoques:
  1. Utilizar vuelta atrás o ramificación y poda: aunque la complejidad en el caso peor sea no polinómica, a menudo son eficientes para muchos casos particulares grandes.
  2. Encontrar un algoritmo eficiente para una subclase de instancias.
  3. Desarrollar un algoritmo aproximado: no se garantiza que devuelva la solución óptima, sino una que esté "razonablemente" cerca de serlo. A menudo podemos encontrar una cota de lo cerca que está la solución devuelta de una óptima.
     Por ejemplo, para el Problema del Viajante podemos encontrar un algoritmo que devuelva una solución con coste *min-aprox* tal que

$$\textit{min-aprox} < 2 \times \textit{min-dist}$$

donde *min-dist* es el coste de una solución óptima.

Algoritmos aproximados

- Sea P un problema, I una instancia de P, y $F^*(I)$ el valor de la solución óptima para I. Un algoritmo aproximado A produce soluciones factibles para I cuyo valor $\hat{F}(I)$ es menor (mayor) que $F^*(I)$ si P es un problema de maximización (minimización).

  Definición: A es un algoritmo aproximado absoluto si para cualquier instancia I

  $$|F^*(I) - \hat{F}(I)| \leq k$$

  para alguna constante k.

  Definición: A tiene una razón de aproximación $\rho(n)$ si, para cualquier instancia I de tamaño n, el coste $\hat{F}(I)$ está en proporción $\rho(n)$ con el coste $F^*(I)$ de una solución óptima:

  $$\forall\, I \text{ de tamaño } n \qquad \max\left(\frac{\hat{F}(I)}{F^*(I)}, \frac{F^*(I)}{\hat{F}(I)}\right) \leq \rho(n).$$

  Un algoritmo que alcanza una razón de aproximación $\rho(n)$ se denomina algoritmo $\rho(n)$-aproximado.

Algoritmos aproximados

- La razón de aproximación nunca es menor que 1, ya que $\frac{\hat{F}(I)}{F^*(I)} < 1$ implica que $\frac{F^*(I)}{\hat{F}(I)} > 1$.
- Un algoritmo 1-aproximado produce soluciones óptimas, y cuanto mayor sea la razón, peores serán las soluciones.

Problema de la cobertura de vértices

- Encontrar la cobertura de menor tamaño.
- Aunque encontrar una solución óptima puede ser difícil, no lo es tanto encontrar una solución que está cerca de ser óptima. El siguiente algoritmo devuelve una cobertura cuyo tamaño no es mayor que el doble del tamaño de una solución óptima.

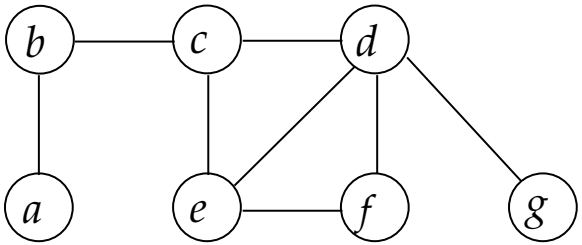
fun `cobertura-aprox`$(G = \langle V, A \rangle)$ **dev** C
  $C := \emptyset$
  $A' := A$
  mientras $A' \neq \emptyset$ **hacer**
    $\langle u, v \rangle :=$ arista arbitraria de A'
    $C := C \cup \{u, v\}$
    eliminar de A' las aristas incidentes en u o v
  fmientras
ffun

- El tiempo de ejecución es $O(|V| + |A|)$.

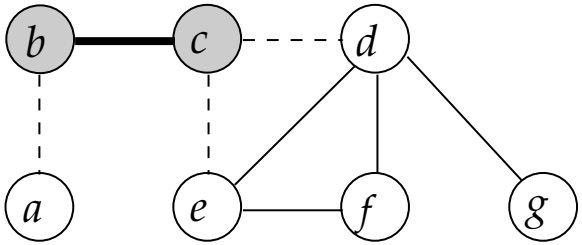
Problema de la cobertura de vértices

Ejemplo:

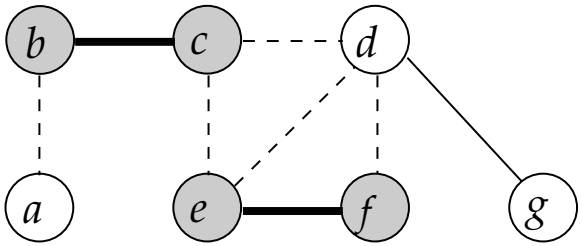
1



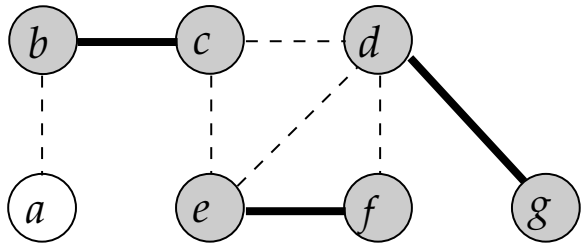
2



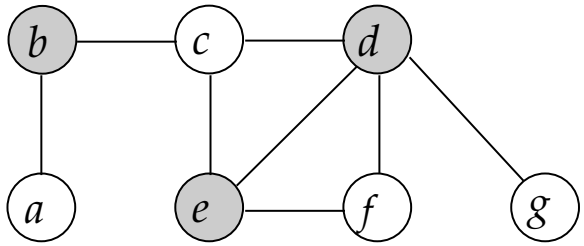
3



4



Solución aproximada



Solución óptima

Teorema 4

El algoritmo `cobertura-aprox` es un algoritmo polinómico 2-aproximado.

Demostración:

- Probemos que la cobertura devuelta tiene como mucho el doble del tamaño de una óptima.
- Sea E el conjunto de aristas seleccionadas por el algoritmo.
- Para cubrir las aristas de E, cualquier cobertura, incluida la óptima C^*, tiene que incluir al menos uno de los extremos de cada arista de E.
- No hay dos aristas en E que compartan algún extremo, ya que cuando se selecciona una arista, todas las aristas incidentes en alguno de sus extremos se elimina de A'.
- Por tanto, no hay dos aristas de E cubiertas por el mismo nodo de C^*, y tenemos la siguiente cota inferior: $|E| \leq |C^*|$.
- Para las aristas seleccionadas, ninguno de sus extremos está cubierto por otras aristas seleccionadas, lo que da una cota superior, de hecho exacta, del tamaño de la cobertura devuelta: $|C| = 2|E|$.
- Por tanto, $|C| \leq 2|C^*|$.

□

Problema del viajante

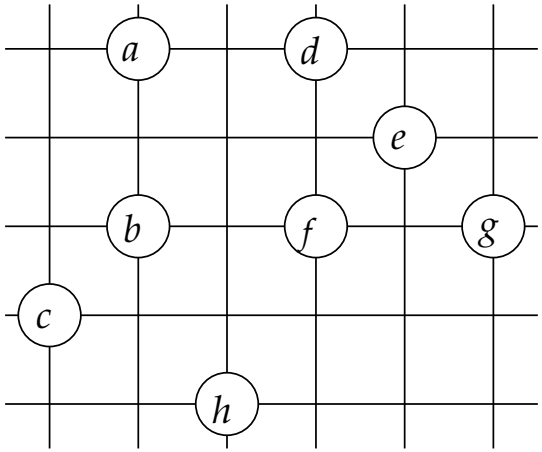
- Nos restringimos a grafos G (no dirigido, completo) que cumplan la desigualdad triangular:
  Para todo vértice $u, v, w \in V$, $\quad G[u,w] \leq G[u,v] + G[v,w]$.
- El problema del viajante sigue siendo $\mathcal{NP}$-difícil aunque se cumpla la desigualdad triangular.
- Procedemos de forma similar al problema anterior:
  1. Calculamos un ARM, cuyo coste resulta ser una cota inferior de la longitud del circuito óptimo.
  2. Utilizamos el ARM para crear un circuito cuyo coste no sea más del doble que el coste del ARM (cumpliéndose la desigualdad triangular).

```
fun circuito-aprox(G) dev H
    T := Prim(G)
    H := preorden(T)
ffun
```

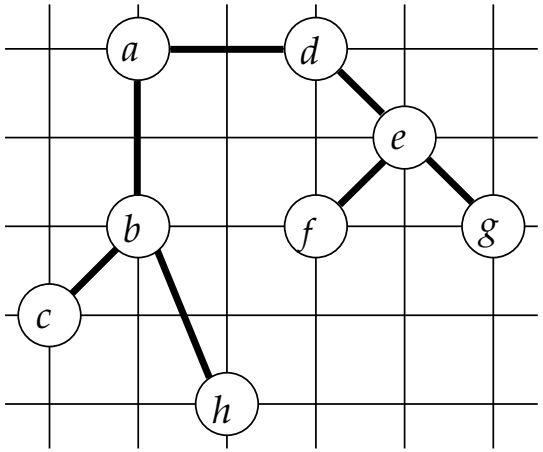
- Complejidad $O(n^2)$.

Problema del viajante

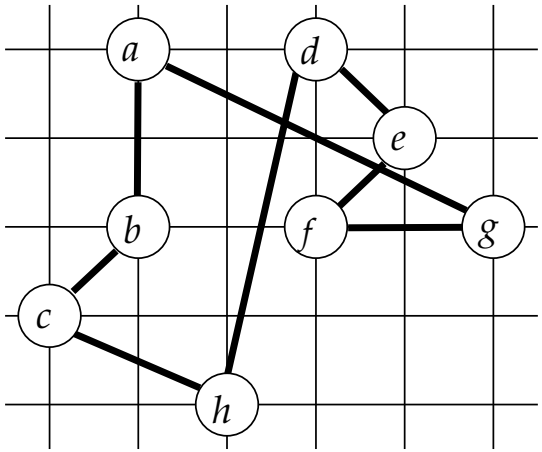
Ejemplo:



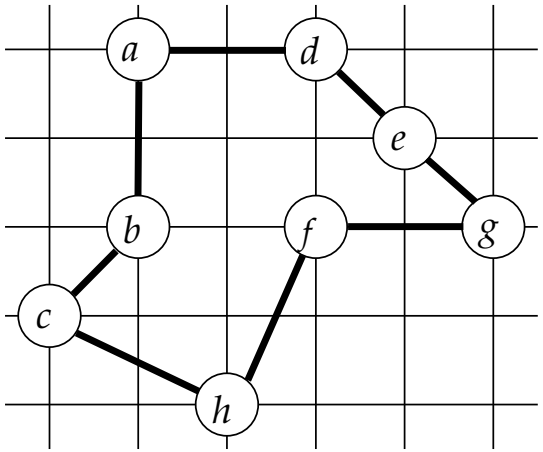
(a)



(b)



(c)



(óptimo)

Teorema 5

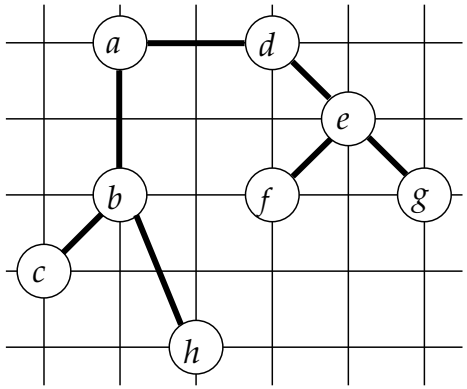
El algoritmo `circuito-aprox` es un algoritmo polinómico 2-aproximado.

Demostración:

- Sea H^* un circuito óptimo.
- Ya que se puede obtener un AR eliminando una arista de un circuito, el coste de un ARM T es cota inferior del coste de H^*,

$$coste(T) \leq coste(H^*).$$

- Un camino completo de T enumera los vértices la primera vez que se visitan y también cuando se vuelve a ellos después de recorrer un subárbol.



- Para el ejemplo, tenemos

$$W = a, b, c, b, h, b, a, d, e, f, e, g, e, d, a.$$

- Como W pasa por cada arista exactamente dos veces, tenemos

$$coste(W) = 2\, coste(T).$$

- Y por tanto,

$$coste(W) \leq 2\, coste(H^*).$$

- W no es un circuito, pero por la desigualdad triangular podemos eliminar cualquier vértice de W y el coste no se incrementa.

- Aplicando repetidamente esta operación podemos dejar solo la primera visita a cada vértice.
  En el ejemplo,

$$W = a,b,c,b,h,b,a,d,e,f,e,g,e,d,a \quad \mapsto \quad H = a,b,c,h,d,e,f,g.$$

- Esta ordenación coincide con el preorden, y es un ciclo Hamiltoniano.
- Como H se obtiene a partir de W eliminando vértices, tenemos

$$coste(H) \leq coste(W).$$

- Y por tanto, $coste(H) \leq 2\, coste(H^*)$.

□

- Hay algoritmos que logran

$$coste(H) \leq 1.5\, coste(H^*) \qquad \text{(ver [NN97, pág. 397]).}$$