Problemas intratables

Elvira Mayordomo

Universidad de Zaragoza

5 de septiembre de 2023

Lectura previa

- Apartado 2 del siguiente artículo (2. WHAT IS THE P VERSUS NP PROBLEM?)
  Lance Fortnow: The Status of the P versus NP problem. Communications of the ACM, 52(9): 78-86. September 2009.
  `https://lance.fortnow.com/papers/files/pnp-cacm.pdf`
- Si tenéis curiosidad podéis echar un vistazo a este segundo artículo.
  Lance Fortnow: Fifty Years of P vs. NP and the Possibility of the Impossible. Communications of the ACM, 65(1): 76-85. January 2022.
  `https://lance.fortnow.com/papers/files/pvnp50.pdf`

Contenido de este tema

Contenido de este tema

Contenido de este tema

1. **Introducción: complejidad y problemas intratables**
2. Reducciones para construir algoritmos
3. Reducciones sencillas para demostrar intratabilidad

Contenido de este tema

1. **Introducción: complejidad y problemas intratables**
2. Reducciones para construir algoritmos
3. Reducciones sencillas para demostrar intratabilidad
4. SAT y 3SAT

Contenido de este tema

1. **Introducción: complejidad y problemas intratables**
2. Reducciones para construir algoritmos
3. Reducciones sencillas para demostrar intratabilidad
4. SAT y 3SAT
5. Complejidad de grano fino: intratabilidad débil

Contenido de este tema

1. **Introducción: complejidad y problemas intratables**
2. Reducciones para construir algoritmos
3. Reducciones sencillas para demostrar intratabilidad
4. SAT y 3SAT
5. Complejidad de grano fino: intratabilidad débil
6. Temas adicionales: NP-completos y NP-difíciles

Contenido de este tema

1. **Introducción: complejidad y problemas intratables**
2. Reducciones para construir algoritmos
3. Reducciones sencillas para demostrar intratabilidad
4. SAT y 3SAT
5. Complejidad de grano fino: intratabilidad débil
6. Temas adicionales: NP-completos y NP-difíciles

Este tema está basado en el capítulo 9 de
Steven S. Skiena. The Algorithm Design Manual. Springer 2008
y en el artículo:
Virginia V. Williams. On some fine-grained questions in algorithms and complexity (ICM 2018)
`https://people.csail.mit.edu/virgi/eccentri.pdf`

Cotas de tiempo para un algoritmo

- ¿Qué quiere decir que un algoritmo tarda tiempo $O(n \log n)$?

Cotas de tiempo para un algoritmo

- ¿Qué quiere decir que un algoritmo tarda tiempo $O(n \log n)$?
- En una cota *superior*, el tiempo que tarda para una entrada de *tamaño* n es menor que $c \cdot n \log n$

Cotas de tiempo para un algoritmo

- ¿Qué quiere decir que un algoritmo tarda tiempo $O(n \log n)$?
- En una cota *superior*, el tiempo que tarda para una entrada de *tamaño* n es menor que $c \cdot n \log n$
- Es tiempo en el *caso peor*, de todas las entradas de tamaño n cogemos la que más tiempo tarda

Cotas de tiempo para un algoritmo

- ¿Qué quiere decir que un algoritmo tarda tiempo $O(n \log n)$?
- En una cota *superior*, el tiempo que tarda para una entrada de *tamaño* n es menor que $c \cdot n \log n$
- Es tiempo en el *caso peor*, de todas las entradas de tamaño n cogemos la que más tiempo tarda
- No tiene que ser una cota ajustada, el tiempo puede ser siempre mucho menor que $c \cdot n \log n$

Cotas de tiempo para un algoritmo

- ¿Qué quiere decir que un algoritmo tarda tiempo $O(n \log n)$?
- En una cota *superior*, el tiempo que tarda para una entrada de *tamaño* n es menor que $c \cdot n \log n$
- Es tiempo en el *caso peor*, de todas las entradas de tamaño n cogemos la que más tiempo tarda
- No tiene que ser una cota ajustada, el tiempo puede ser siempre mucho menor que $c \cdot n \log n$
- Cuanto más ajustada sea la cota superior más útil

Cotas de tiempo para un algoritmo

Cotas de tiempo para un algoritmo

- ¿Y cotas inferiores?

Cotas de tiempo para un algoritmo

- ¿Y cotas inferiores?
- **Para un algoritmo** suele ser posible decir que tarda tiempo $\Omega(t(n))$, es decir, tiempo mayor que $c \cdot t(n)$ en el caso peor

Cotas de tiempo para un algoritmo

- ¿Y cotas inferiores?
- **Para un algoritmo** suele ser posible decir que tarda tiempo $\Omega(t(n))$, es decir, tiempo mayor que $c \cdot t(n)$ en el caso peor
- Tiempo mayor que $c \cdot t(n)$ en el caso peor: hay una c tal que para cada n al menos una entrada de tamaño n tarda más de $c \cdot t(n)$

Cotas de tiempo para un problema

- Buscamos siempre el *mejor* algoritmo para un problema

Cotas de tiempo para un problema

- Buscamos siempre el *mejor* algoritmo para un problema
- Cuando lo que nos interesa mejorar es el tiempo vamos a buscar el algoritmo con *menor tiempo* en caso peor

Cotas de tiempo para un problema

- Buscamos siempre el *mejor* algoritmo para un problema
- Cuando lo que nos interesa mejorar es el tiempo vamos a buscar el algoritmo con *menor tiempo* en caso peor
- **Cotas superiores:** Si encontramos un algoritmo que resuelve el problema A en tiempo $O(t(n))$ decimos que el problema está en $\mathrm{DTIME}(t(n))$

Cotas de tiempo para un problema

- Buscamos siempre el *mejor* algoritmo para un problema
- Cuando lo que nos interesa mejorar es el tiempo vamos a buscar el algoritmo con *menor tiempo* en caso peor
- **Cotas superiores:** Si encontramos un algoritmo que resuelve el problema A en tiempo $O(t(n))$ decimos que el problema está en $\mathrm{DTIME}(t(n))$
- Otra vez, esto puede ser muy poco ajustado ...

Cotas de tiempo para un problema

- Buscamos siempre el *mejor* algoritmo para un problema
- Cuando lo que nos interesa mejorar es el tiempo vamos a buscar el algoritmo con *menor tiempo* en caso peor
- **Cotas superiores:** Si encontramos un algoritmo que resuelve el problema A en tiempo $O(t(n))$ decimos que el problema está en $\mathrm{DTIME}(t(n))$
- Otra vez, esto puede ser muy poco ajustado ...
- El algoritmo puede tardar siempre mucho menos tiempo que $c \cdot t(n)$

Cotas de tiempo para un problema

- Buscamos siempre el *mejor* algoritmo para un problema
- Cuando lo que nos interesa mejorar es el tiempo vamos a buscar el algoritmo con *menor tiempo* en caso peor
- **Cotas superiores:** Si encontramos un algoritmo que resuelve el problema A en tiempo $O(t(n))$ decimos que el problema está en $\mathrm{DTIME}(t(n))$
- Otra vez, esto puede ser muy poco ajustado ...
- El algoritmo puede tardar siempre mucho menos tiempo que $c \cdot t(n)$
- Más importante: **puede haber otros algoritmos que resuelvan el problema mucho más rápido**

Cotas de tiempo para un problema

Cotas de tiempo para un problema

- **Es muy raro encontrar cotas inferiores para el tiempo de resolución de un problema** ¿Conoces alguna?

Cotas de tiempo para un problema

- **Es muy raro encontrar cotas inferiores para el tiempo de resolución de un problema** ¿Conoces alguna?
- Ordenación es uno de los pocos problemas que tiene cotas muy ajustadas

¿Qué es un problema computacionalmente intratable?

¿Qué es un problema computacionalmente intratable?

- **Un problema intratable es un problema que ningún algoritmo resuelve suficientemente rápido**

¿Qué es un problema computacionalmente intratable?

- **Un problema intratable es un problema que ningún algoritmo resuelve suficientemente rápido**
- ¿qué es suficientemente rápido?

¿Qué es un problema computacionalmente intratable?

- **Un problema intratable es un problema que ningún algoritmo resuelve suficientemente rápido**
- ¿qué es suficientemente rápido?
- Acabas de decir que no sabemos casi ninguna cota inferior de tiempo para problemas

¿Qué es un problema computacionalmente intratable?

- **Un problema intratable es un problema que ningún algoritmo resuelve suficientemente rápido**
- ¿qué es suficientemente rápido?
- Acabas de decir que no sabemos casi ninguna cota inferior de tiempo para problemas
- Diremos que un problema es intratable **mientras no encontremos ningún algoritmo que lo resuelva suficientemente rápido**

¿Qué es un problema computacionalmente intratable?

- **Un problema intratable es un problema que ningún algoritmo resuelve suficientemente rápido**
- ¿qué es suficientemente rápido?
- Acabas de decir que no sabemos casi ninguna cota inferior de tiempo para problemas
- Diremos que un problema es intratable **mientras no encontremos ningún algoritmo que lo resuelva suficientemente rápido**
- En general los problemas intratables:
  - tienen algoritmos de fuerza bruta o sencillos pero muy lentos

¿Qué es un problema computacionalmente intratable?

- **Un problema intratable es un problema que ningún algoritmo resuelve suficientemente rápido**
- ¿qué es suficientemente rápido?
- Acabas de decir que no sabemos casi ninguna cota inferior de tiempo para problemas
- Diremos que un problema es intratable **mientras no encontremos ningún algoritmo que lo resuelva suficientemente rápido**
- En general los problemas intratables:
  - tienen algoritmos de fuerza bruta o sencillos pero muy lentos
  - hace décadas que no se han conseguido algoritmos más rápidos

Un par de ejemplos de intratables

SAT-SQL

Problema: SAT-SQL
Entrada: Una tabla T de n atributos y m tuplas y una consulta SQL q sobre esa tabla
Salida: ¿devuelve q algún resultado sobre T?

SAT-SQL

Problema: SAT-SQL
Entrada: Una tabla T de n atributos y m tuplas y una consulta SQL q sobre esa tabla
Salida: ¿devuelve q algún resultado sobre T?

- ¿Cuánto tiempo cuesta resolver SAT-SQL?

SAT-SQL

Problema: SAT-SQL
Entrada: Una tabla T de n atributos y m tuplas y una consulta SQL q sobre esa tabla
Salida: ¿devuelve q algún resultado sobre T?

- ¿Cuánto tiempo cuesta resolver SAT-SQL?
- Aun restringiendo mucho la cantidad de valores distintos de los atributos, los mejores algoritmos conocidos tardan más de 2^n

SAT-SQL

Problema: SAT-SQL
Entrada: Una tabla T de n atributos y m tuplas y una consulta SQL q sobre esa tabla
Salida: ¿devuelve q algún resultado sobre T?

- ¿Cuánto tiempo cuesta resolver SAT-SQL?
- Aun restringiendo mucho la cantidad de valores distintos de los atributos, los mejores algoritmos conocidos tardan más de 2^n
- ¿Y todas esas clases sobre normalización de tablas, optimización de consultas?

Longest Common Subsequence

Problema: LCS
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

Longest Common Subsequence

Problema: LCS
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

- ¿Cuánto tiempo cuesta resolver LCS?

Longest Common Subsequence

Problema: LCS
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

- ¿Cuánto tiempo cuesta resolver LCS?
- Con programación dinámica se puede hacer en tiempo $O(n^2)$

Longest Common Subsequence

Problema: LCS
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

- ¿Cuánto tiempo cuesta resolver LCS?
- Con programación dinámica se puede hacer en tiempo $O(n^2)$
- El mejor algoritmo conocido tarda $O(n^2/\log^2 n)$

Longest Common Subsequence

Problema: LCS
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

- ¿Cuánto tiempo cuesta resolver LCS?
- Con programación dinámica se puede hacer en tiempo $O(n^2)$
- El mejor algoritmo conocido tarda $O(n^2/\log^2 n)$
- Está bien, ¿no?

Longest Common Subsequence

Problema: LCS
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

- ¿Cuánto tiempo cuesta resolver LCS?
- Con programación dinámica se puede hacer en tiempo $O(n^2)$
- El mejor algoritmo conocido tarda $O(n^2/\log^2 n)$
- Está bien, ¿no?
- Se usa extensivamente en biología molecular y en correctores ortográficos ...

Longest Common Subsequence

Problema: LCS
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

- ¿Cuánto tiempo cuesta resolver LCS?
- Con programación dinámica se puede hacer en tiempo $O(n^2)$
- El mejor algoritmo conocido tarda $O(n^2/\log^2 n)$
- Está bien, ¿no?
- Se usa extensivamente en biología molecular y en correctores ortográficos ...
- Por ejemplo se le puede dar como datos dos cadenas de ADN de 1Mbp (10^6 letras) ...

Tiempo polinómico

- Tiempo polinómico es tiempo $O(n^k)$ para algún k

Tiempo polinómico

- Tiempo polinómico es tiempo $O(n^k)$ para algún k
- Tiempo $O(n)$, $O(n^2)$, $O(n^3)$, ...

Tiempo polinómico

- Tiempo polinómico es tiempo $O(n^k)$ para algún k
- Tiempo $O(n)$, $O(n^2)$, $O(n^3)$, ...
- ¿Es tiempo polinómico eficiente?
- ¿$O(n^{100})$?

Tiempo polinómico

- Tiempo polinómico es tiempo $O(n^k)$ para algún k
- Tiempo $O(n)$, $O(n^2)$, $O(n^3)$, ...
- ¿Es tiempo polinómico eficiente?
- ¿$O(n^{100})$?
- La versión clásica es que los problemas interesantes en tiempo polinómico están en $\mathrm{DTIME}(n^k)$ para k pequeño (como mucho 3 ó 4 el algún caso raro)
- ¿Es tiempo polinómico eficiente?

Tiempo polinómico

- Tiempo polinómico es tiempo $O(n^k)$ para algún k
- Tiempo $O(n)$, $O(n^2)$, $O(n^3)$, ...
- ¿Es tiempo polinómico eficiente?
- ¿$O(n^{100})$?
- La versión clásica es que los problemas interesantes en tiempo polinómico están en $\mathrm{DTIME}(n^k)$ para k pequeño (como mucho 3 ó 4 el algún caso raro)
- ¿Es tiempo polinómico eficiente?
- En la época de los datos masivos, cuadrático ($k = 2$) ya puede ser demasiado

Tiempo polinómico

- Tiempo polinómico es tiempo $O(n^k)$ para algún k
- Tiempo $O(n)$, $O(n^2)$, $O(n^3)$, ...
- ¿Es tiempo polinómico eficiente?
- ¿$O(n^{100})$?
- La versión clásica es que los problemas interesantes en tiempo polinómico están en $\mathrm{DTIME}(n^k)$ para k pequeño (como mucho 3 ó 4 el algún caso raro)
- ¿Es tiempo polinómico eficiente?
- En la época de los datos masivos, cuadrático ($k = 2$) ya puede ser demasiado

Tiempo polinómico = eficiente (versión "clásica")

Tiempo cuasilineal

- Si al menos tienes que leer los datos de entrada completos lo mínimo que necesitas es *tiempo lineal* n (donde n es el tamaño de los datos de entrada)

Tiempo cuasilineal

- Si al menos tienes que leer los datos de entrada completos lo mínimo que necesitas es *tiempo lineal* n (donde n es el tamaño de los datos de entrada)
- En realidad para no hacer trampas con el acceso directo a distintas partes de los datos es mejor decir $n \log n$ (vamos, que el tiempo lineal no existe si lees los datos completos)

Tiempo cuasilineal

- Si al menos tienes que leer los datos de entrada completos lo mínimo que necesitas es *tiempo lineal* n (donde n es el tamaño de los datos de entrada)
- En realidad para no hacer trampas con el acceso directo a distintas partes de los datos es mejor decir $n \log n$ (vamos, que el tiempo lineal no existe si lees los datos completos)
- No nos preocupan mucho los factores logarítmicos así que cualquier cota $n \log^k n$ será equivalente

Tiempo cuasilineal

- Si al menos tienes que leer los datos de entrada completos lo mínimo que necesitas es *tiempo lineal* n (donde n es el tamaño de los datos de entrada)
- En realidad para no hacer trampas con el acceso directo a distintas partes de los datos es mejor decir $n \log n$ (vamos, que el tiempo lineal no existe si lees los datos completos)
- No nos preocupan mucho los factores logarítmicos así que cualquier cota $n \log^k n$ será equivalente
- *Tiempo cuasilineal* es tiempo $O(n \log^k n)$ (también escrito $O(n \operatorname{polylog}(n))$)

Tiempo cuasilineal

- Si al menos tienes que leer los datos de entrada completos lo mínimo que necesitas es *tiempo lineal* n (donde n es el tamaño de los datos de entrada)
- En realidad para no hacer trampas con el acceso directo a distintas partes de los datos es mejor decir $n \log n$ (vamos, que el tiempo lineal no existe si lees los datos completos)
- No nos preocupan mucho los factores logarítmicos así que cualquier cota $n \log^k n$ será equivalente
- *Tiempo cuasilineal* es tiempo $O(n \log^k n)$ (también escrito $O(n \operatorname{polylog}(n))$)

Tiempo cuasilineal = eficiente (era del big data)

Intratabilidad de SAT-SQL

Hipótesis de trabajo 1

No hay ningún algoritmo que resuelva SAT-SQL en tiempo polinómico

Intratabilidad débil de LCS

Hipótesis de trabajo 2

No hay ningún algoritmo que resuelva LCS en tiempo cuasilineal

Dogma de fé

Toda esta asignatura, buena parte de la informática, la criptografía, etc se hacen partiendo de que SAT-SQL es intratable

Algo similar ocurre con que LCS es intratable débil

Resumen

- Hemos repasado las cotas de complejidad en tiempo de un algoritmo y de un problema
- Un problema intratable es aquel para el que **no se conocen** algoritmos rápidos
- Dependiendo del contexto "algoritmo rápido" es aquel que tarda tiempo polinómico o bien tiempo cuasilineal

Contenido de este tema

1. Introducción: complejidad y problemas intratables
2. **Reducciones para construir algoritmos**
3. Reducciones sencillas para demostrar intratabilidad
4. SAT y 3SAT
5. P vs NP
6. Complejidad de grano fino: intratabilidad débil
7. Temas adicionales: NP-completos y NP-difíciles

Introducción: Reducciones

- Un problema intratable es aquel para el que **no se conocen** algoritmos rápidos

Introducción: Reducciones

- Un problema intratable es aquel para el que **no se conocen** algoritmos rápidos
- ¿Y eso cómo se decide en la práctica?

Introducción: Reducciones

- Un problema intratable es aquel para el que **no se conocen** algoritmos rápidos
- ¿Y eso cómo se decide en la práctica?
- El concepto fundamental es el de reducción entre dos problemas, que permite **compararlos**

Introducción: Reducciones

- Un problema intratable es aquel para el que **no se conocen** algoritmos rápidos
- ¿Y eso cómo se decide en la práctica?
- El concepto fundamental es el de reducción entre dos problemas, que permite **compararlos**
- Método a usar: Conozco unos cuantos problemas intratables y comparo los nuevos problemas con ellos usando reducciones

Introducción: Reducciones

- Un problema intratable es aquel para el que **no se conocen** algoritmos rápidos
- ¿Y eso cómo se decide en la práctica?
- El concepto fundamental es el de reducción entre dos problemas, que permite **compararlos**
- Método a usar: Conozco unos cuantos problemas intratables y comparo los nuevos problemas con ellos usando reducciones
- Si A es reducible a B y tenemos un algoritmo eficiente para B entonces tenemos un algoritmo eficiente para A.
- Si A es reducible a B y no existe un algoritmo eficiente para A entonces no existe un algoritmo eficiente para B

$$A \leq B$$

Introducción: Reducciones

- Un problema intratable es aquel para el que **no se conocen** algoritmos rápidos
- ¿Y eso cómo se decide en la práctica?
- El concepto fundamental es el de reducción entre dos problemas, que permite **compararlos**
- Método a usar: Conozco unos cuantos problemas intratables y comparo los nuevos problemas con ellos usando reducciones
- Si A es reducible a B y tenemos un algoritmo eficiente para B entonces tenemos un algoritmo eficiente para A.
- Si A es reducible a B y no existe un algoritmo eficiente para A entonces no existe un algoritmo eficiente para B

$$A \leq B$$

Introducción: Reducciones

- Un problema intratable es aquel para el que **no se conocen** algoritmos rápidos
- ¿Y eso cómo se decide en la práctica?
- El concepto fundamental es el de reducción entre dos problemas, que permite **compararlos**
- Método a usar: Conozco unos cuantos problemas intratables y comparo los nuevos problemas con ellos usando reducciones
- Si A es reducible a B y tenemos un algoritmo eficiente para B entonces tenemos un algoritmo eficiente para A.
- Si A es reducible a B y no existe un algoritmo eficiente para A entonces no existe un algoritmo eficiente para B

$$A \leq B$$

Mencionado en el tema *7-Programación lineal y reducciones* de la asignatura de *Algoritmia Básica*

Reducciones para construir algoritmos

- ¿Cómo usar reducciones para hacer algoritmos?

Reducciones para construir algoritmos

- ¿Cómo usar reducciones para hacer algoritmos?
- Usamos algoritmos conocidos para construir otros nuevos.
- Si podemos traducir un problema nuevo A que queremos resolver a uno B que ya sabemos resolver tenemos un algoritmo para resolver el problema nuevo.

Reducciones para construir algoritmos

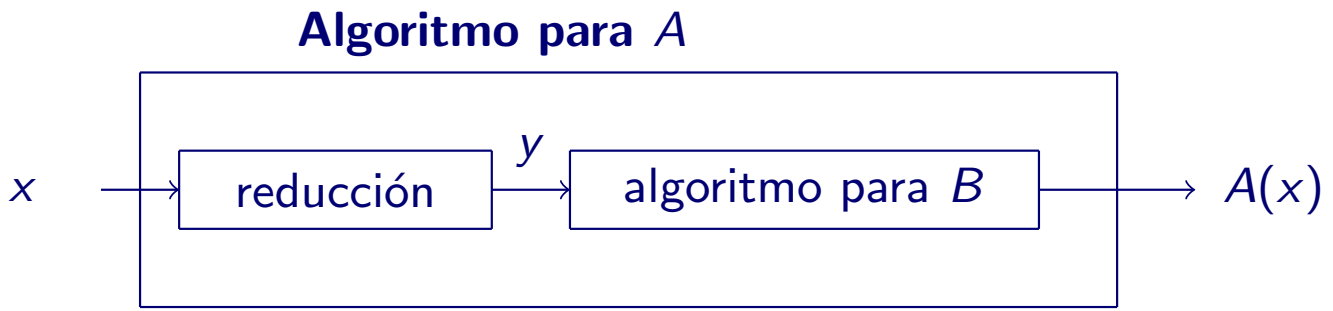
- ¿Cómo usar reducciones para hacer algoritmos?
- Usamos algoritmos conocidos para construir otros nuevos.
- Si podemos traducir un problema nuevo A que queremos resolver a uno B que ya sabemos resolver tenemos un algoritmo para resolver el problema nuevo.
- ¿Cómo? Usamos primero la reducción del nuevo al viejo y después el algoritmo para el viejo.

Reducciones para construir algoritmos

- ¿Cómo usar reducciones para hacer algoritmos?
- Usamos algoritmos conocidos para construir otros nuevos.
- Si podemos traducir un problema nuevo A que queremos resolver a uno B que ya sabemos resolver tenemos un algoritmo para resolver el problema nuevo.
- ¿Cómo? Usamos primero la reducción del nuevo al viejo y después el algoritmo para el viejo.
- El coste en tiempo será la suma del tiempo de la reducción más el tiempo del algoritmo para el viejo.

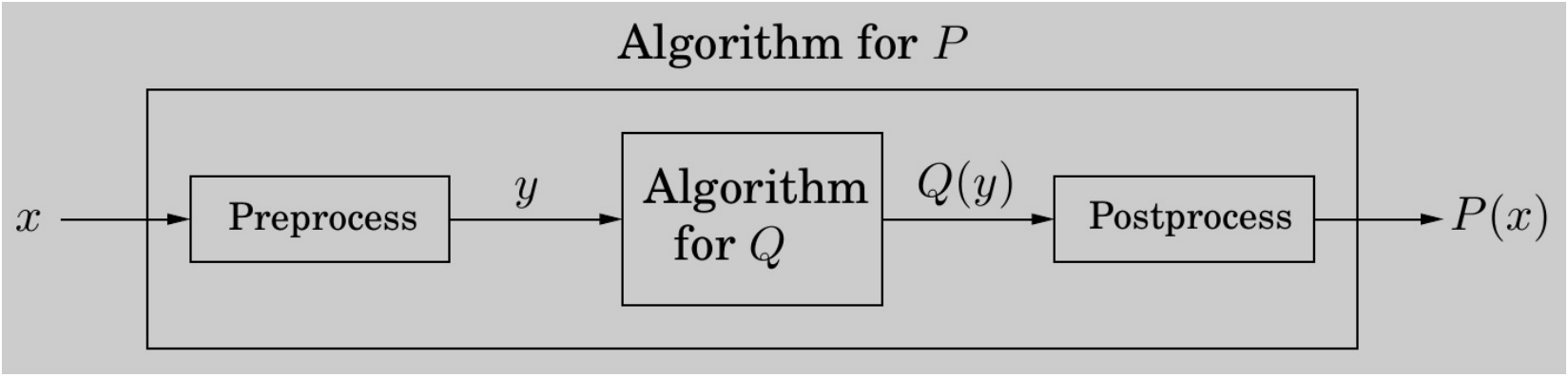
Reducciones para construir algoritmos

- ¿Cómo usar reducciones para hacer algoritmos?
- Usamos algoritmos conocidos para construir otros nuevos.
- Si podemos traducir un problema nuevo A que queremos resolver a uno B que ya sabemos resolver tenemos un algoritmo para resolver el problema nuevo.
- ¿Cómo? Usamos primero la reducción del nuevo al viejo y después el algoritmo para el viejo.
- El coste en tiempo será la suma del tiempo de la reducción más el tiempo del algoritmo para el viejo.



Reducciones para construir algoritmos

En realidad una reducción puede ser más complicada, veamos lo visto en Algoritmia Básica: una reducción de P a Q:



Reducciones para construir algoritmos

Todavía puede ser más complicada: una reducción de A a B puede usar varios casos de B:

$\mathrm{A}(x)$

1 **for** $i = 1$ **to** k
2 Calcula y_i (a partir de $z_1, \ldots, z_{i-1}$)
3 Calcula $z_i = B(y_i)$
4 Cálculos finales a partir de $z_1, \ldots, z_k$
5 Resultado $\mathrm{A}(x)$

Reducciones para construir algoritmos

Todavía puede ser más complicada: una reducción de A a B puede usar varios casos de B:

$\mathrm{A}(x)$

1 **for** $i = 1$ **to** k
2 Calcula y_i (a partir de $z_1, \ldots, z_{i-1}$)
3 Calcula $z_i = B(y_i)$
4 Cálculos finales a partir de $z_1, \ldots, z_k$
5 Resultado $\mathrm{A}(x)$

- El tiempo es $\sum_i \mathrm{tiempo}(B(y_i))$ más los cálculos 2 y 4

Subsecuencia común más larga

Problema: Longest Common Subsequence (LCS)
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

Subsecuencia común más larga

Problema: Longest Common Subsequence (LCS)
Entrada: Dos cadenas con n letras cada una
Salida: Cadena que es subsecuencia de las dos (no necesariamente contigua) de longitud máxima

```
ATCGGGTTCCTTAAGGG
ATTGGTACCTTCAGGCC
```

¿A qué problema os recuerda de los que visteis en AB?

Subsecuencia común más larga

En AB visteis técnicas de programación dinámica para calcular la distancia de edición (comparaciones de secuencias)

Problema: Distancia de edición
Entrada: Cadenas de caracteres o enteros S y T; coste de cada inserción (c_{ins}), borrado (c_{del}), y sustitución (c_{sub})
Salida: ¿Cuál es el coste mínimo de las operaciones para transformar S en T?

Subsecuencia común más larga

En AB visteis técnicas de programación dinámica para calcular la distancia de edición (comparaciones de secuencias)

Problema: Distancia de edición
Entrada: Cadenas de caracteres o enteros S y T; coste de cada inserción (c_{ins}), borrado (c_{del}), y sustitución (c_{sub})
Salida: ¿Cuál es el coste mínimo de las operaciones para transformar S en T?

Problema: Longitud de subsecuencia común más larga (LLCS)
Entrada: Dos cadenas con n letras cada una
Salida: ¿Cuál es la longitud de la subsecuencia común a las dos (no necesariamente contigua) de longitud máxima?

Reducciones para construir algoritmos

De hecho *Longitud de subsecuencia común más larga* se puede reducir a *Distancia de edición*...

Reducciones para construir algoritmos

De hecho *Longitud de subsecuencia común más larga* se puede reducir a *Distancia de edición*...

$\mathrm{LLCS}(S, T)$

1 $c_{ins} = c_{del} = 1$
2 $c_{sub} = \infty$
3 Resultado $|S|-\text{distanciaDeEdición}(S, T, c_{ins}, c_{del}, c_{sub})/2$

Reducciones para construir algoritmos

De hecho *Longitud de subsecuencia común más larga* se puede reducir a *Distancia de edición*...

$\mathrm{LLCS}(S, T)$

1 $c_{ins} = c_{del} = 1$
2 $c_{sub} = \infty$
3 Resultado $|S|-\text{distanciaDeEdición}(S, T, c_{ins}, c_{del}, c_{sub})/2$

¿Por qué funciona?

Reducciones para construir algoritmos

De hecho *Longitud de subsecuencia común más larga* se puede reducir a *Distancia de edición*...

$\mathrm{LLCS}(S, T)$

1 $c_{ins} = c_{del} = 1$
2 $c_{sub} = \infty$
3 Resultado $|S|-\text{distanciaDeEdición}(S, T, c_{ins}, c_{del}, c_{sub})/2$

¿Por qué funciona?

Reducciones para construir algoritmos

De hecho *Longitud de subsecuencia común más larga* se puede reducir a *Distancia de edición*...

$\mathrm{LLCS}(S, T)$

1 $c_{ins} = c_{del} = 1$
2 $c_{sub} = \infty$
3 Resultado $|S|-\text{distanciaDeEdición}(S, T, c_{ins}, c_{del}, c_{sub})/2$

¿Por qué funciona?

- No permitimos sustituciones ($c_{sub} = \infty$) luego la edición óptima deja la subsecuencia común más larga, borra el resto de S e inserta el resto de T (cada inserción y cada borrado con coste 1)

Reducciones para construir algoritmos

De hecho *Longitud de subsecuencia común más larga* se puede reducir a *Distancia de edición*...

$\mathrm{LLCS}(S, T)$

1 $c_{ins} = c_{del} = 1$
2 $c_{sub} = \infty$
3 Resultado $|S|-\text{distanciaDeEdición}(S, T, c_{ins}, c_{del}, c_{sub})/2$

¿Por qué funciona?

- No permitimos sustituciones ($c_{sub} = \infty$) luego la edición óptima deja la subsecuencia común más larga, borra el resto de S e inserta el resto de T (cada inserción y cada borrado con coste 1)
- El coste dividido por 2 es el número de elementos fuera de la LCS

Subsecuencia común más larga

Subsecuencia común más larga

¿Cuánto cuesta el algoritmo para LLCS usando DistanciaDeEdición?

Subsecuencia común más larga

¿Cuánto cuesta el algoritmo para LLCS usando DistanciaDeEdición?
El coste es el de distancia de edición $O(|S| \cdot |T|) = O(n^2)$ más constante

Subsecuencia común más larga

¿Cuánto cuesta el algoritmo para LLCS usando DistanciaDeEdición?
El coste es el de distancia de edición $O(|S| \cdot |T|) = O(n^2)$ más constante
¿Qué podemos deducir de que LLCS es débilmente intratable (es decir, (hay fuertes sospechas de que) no hay ningún algoritmo que resuelve LLCS en tiempo cuasilineal)?

Ejercicio

Problema: Edición
Entrada: Cadenas de caracteres o enteros S y T; coste de cada inserción (c_{ins}), borrado (c_{del}), y sustitución (c_{sub})
Salida: ¿Cuál es la secuencia de operaciones para transformar S en T con coste mínimo?

Ejercicio: Reducir LCS a Edición

Mínimo común múltiplo

Decimos que a divide a b ($b|a$) si existe un entero d tal que $a = bd$

Problema: m.c.m.
Entrada: Enteros x, y
Salida: ¿Cuál es el menor entero m tal que m es múltiplo de x y m es múltiplo de y ?

Problema: m.c.d.
Entrada: Enteros x, y
Salida: ¿Cuál es el mayor entero d tal que d divide a x y d divide a y ?

m.c.m.(24,36)=72 y m.c.d.(24,36)=12

Mínimo común múltiplo

- Se pueden resolver hallando los factores primos de x, y pero no se conocen algoritmos eficientes para factorizar

Mínimo común múltiplo

- Se pueden resolver hallando los factores primos de x, y pero no se conocen algoritmos eficientes para factorizar
- El algoritmo de Euclides es una forma eficiente de hallar el m.c.d. ($O(n)$) (donde n es la suma del número de bits)

Mínimo común múltiplo

- Se pueden resolver hallando los factores primos de x, y pero no se conocen algoritmos eficientes para factorizar
- El algoritmo de Euclides es una forma eficiente de hallar el m.c.d. ($O(n)$) (donde n es la suma del número de bits)
- El algoritmo eficiente para m.c.m. pasa por una reducción a m.c.d.

$\text{M.C.M.}(x, y)$

1 Resultado $xy/m.c.d.(x, y)$

Reducciones para construir algoritmos: resumiendo

- Si reduzco A a B tengo un algoritmo para A que cuesta el tiempo de la reducción más el tiempo de B
  - Es el diseño descendente de toda la vida
- Si reduzco A a B puedo conseguir una cota inferior para B
  - Hemos reducido *LCS* a *DistanciaDeEdición*. Si suponemos que LCS tiene cota inferior $\Omega(n^2/\log^2 n)$ tenemos una cota inferior para DistanciaDeEdición

Contenido de este tema

1. Introducción: complejidad y problemas intratables
2. Reducciones para construir algoritmos
3. **Reducciones sencillas para demostrar intratabilidad**
4. SAT y 3SAT
5. P vs NP
6. Complejidad de grano fino: intratabilidad débil
7. Temas adicionales: NP-completos y NP-difíciles

Reducciones para comparar problemas

Hemos visto muchos problemas en Algoritmia Básica para los que no hemos encontrado algoritmos que tarden tiempo polinómico:

- mochila 0-1
- viajante de comercio
- juego del 15
- planificación de tareas con plazo fijo
- graph coloring
- hamiltonian cycles
- ajedrez
- ...

¿Cómo nos ayudan las reducciones en estos casos?

Reducciones para comparar problemas

Una pelea de niños:

- Pedro vence a Juan, y Juan vence a Chuck
- ¿Quién es fuerte?

Reducciones para comparar problemas

Una pelea de niños:

- Pedro vence a Juan, y Juan vence a Chuck
- ¿Quién es fuerte?
- ¿Y si Chuck es Chuck Norris?

Reducciones para comparar problemas

Una pelea de niños:

- Pedro vence a Juan, y Juan vence a Chuck
- ¿Quién es fuerte?
- ¿Y si Chuck es Chuck Norris? Entonces Juan y Pedro son tan duros como Chuck Norris.

Reducciones para comparar problemas

Una pelea de niños:

- Pedro vence a Juan, y Juan vence a Chuck
- ¿Quién es fuerte?
- ¿Y si Chuck es Chuck Norris? Entonces Juan y Pedro son tan duros como Chuck Norris.
- ¿Y si es el patio de la guardería?

Reducciones para comparar problemas

Una pelea de niños:

- Pedro vence a Juan, y Juan vence a Chuck
- ¿Quién es fuerte?
- ¿Y si Chuck es Chuck Norris? Entonces Juan y Pedro son tan duros como Chuck Norris.
- ¿Y si es el patio de la guardería?

$Chuck \leq Juan \leq Pedro$

Reducciones sencillas para intratabilidad

- Queremos demostrar que algunos problemas son intratables, es decir, que no tienen algoritmos que los resuelvan en tiempo polinómico

[1] Después justificamos porqué estos dos son intratables

Reducciones sencillas para intratabilidad

- Queremos demostrar que algunos problemas son intratables, es decir, que no tienen algoritmos que los resuelvan en tiempo polinómico
- Partimos de dos intratables: *Ciclo Hamiltoniano* y *Cobertura de Vértices*[1].

[1] Después justificamos porqué estos dos son intratables

Reducciones sencillas para intratabilidad

- Queremos demostrar que algunos problemas son intratables, es decir, que no tienen algoritmos que los resuelvan en tiempo polinómico
- Partimos de dos intratables: *Ciclo Hamiltoniano* y *Cobertura de Vértices*[1].

[1] Después justificamos porqué estos dos son intratables

Reducciones sencillas para intratabilidad

- Queremos demostrar que algunos problemas son intratables, es decir, que no tienen algoritmos que los resuelvan en tiempo polinómico
- Partimos de dos intratables: *Ciclo Hamiltoniano* y *Cobertura de Vértices*[1].

- Si *Ciclo Hamiltoniano* es reducible a B entonces B es intratable
- Si *Cobertura de Vértices* es reducible a B entonces B es intratable

[1] Después justificamos porqué estos dos son intratables

Reducciones sencillas para intratabilidad

- Queremos demostrar que algunos problemas son intratables, es decir, que no tienen algoritmos que los resuelvan en tiempo polinómico
- Partimos de dos intratables: *Ciclo Hamiltoniano* y *Cobertura de Vértices*[1].

- Si *Ciclo Hamiltoniano* es reducible a B entonces B es intratable
- Si *Cobertura de Vértices* es reducible a B entonces B es intratable

$$\begin{aligned} \mathit{CicloHam} &\leq B \\ \mathit{CoberturaV} &\leq B \end{aligned}$$

"B vence a Chuck"

[1] Después justificamos porqué estos dos son intratables

Definición de Ciclo Hamiltoniano

Problema: Ciclo Hamiltoniano
Entrada: Un grafo G.
Salida: ¿Existe un ciclo hamiltoniano en G, es decir, un camino que visita una vez cada vértice y vuelve al vértice inicial?

Definición de Cobertura de Vértices

Problema: Cobertura de Vértices
Entrada: Un grafo G (con vértices V) y $k \in \mathbb{N}$.
Salida: ¿Existe un conjunto U de k vértices de G tal que cada arista (i, j) de G cumple que $i \in U$ ó $j \in U$?

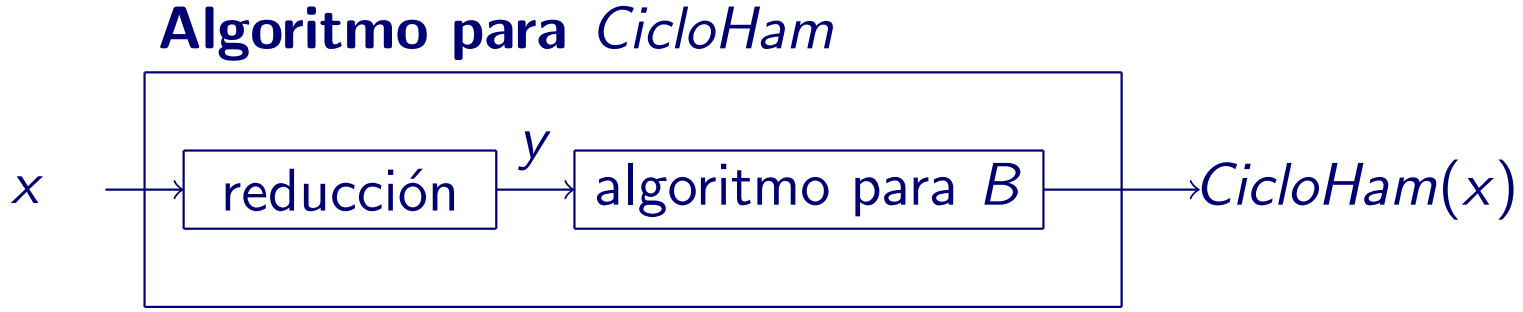
Definición de Cobertura de Vértices

Problema: Cobertura de Vértices
Entrada: Un grafo G (con vértices V) y $k \in \mathbb{N}$.
Salida: ¿Existe un conjunto U de k vértices de G tal que cada arista (i, j) de G cumple que $i \in U$ ó $j \in U$?

- Cuanto más pequeño k más difícil es la Cobertura de Vértices

Reducción de *Ciclo Hamiltoniano* a B

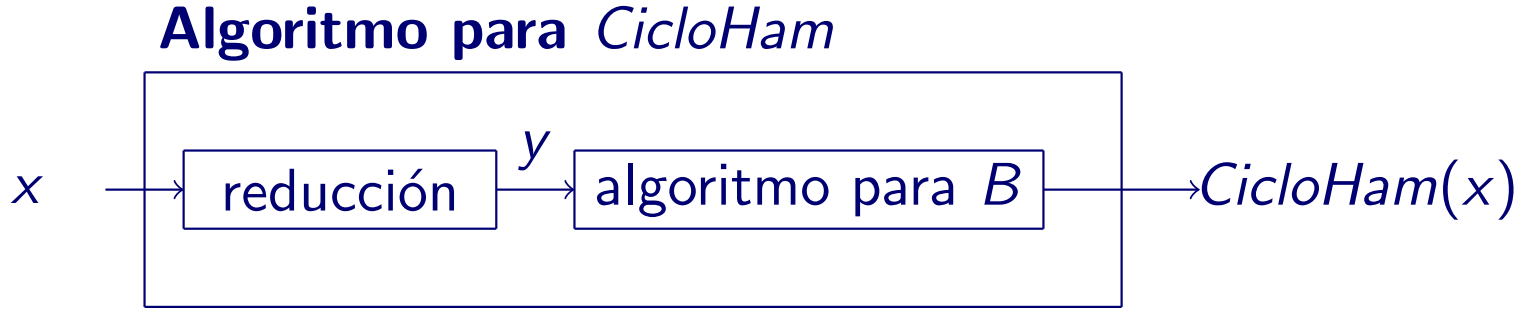
$$\mathit{CicloHam} \leq B$$



- Transformamos la entrada de *Ciclo Hamiltoniano* en entrada de B **en tiempo polinómico**
- La solución de B nos da la solución para *Ciclo Hamiltoniano*

Reducción de *Ciclo Hamiltoniano* a B

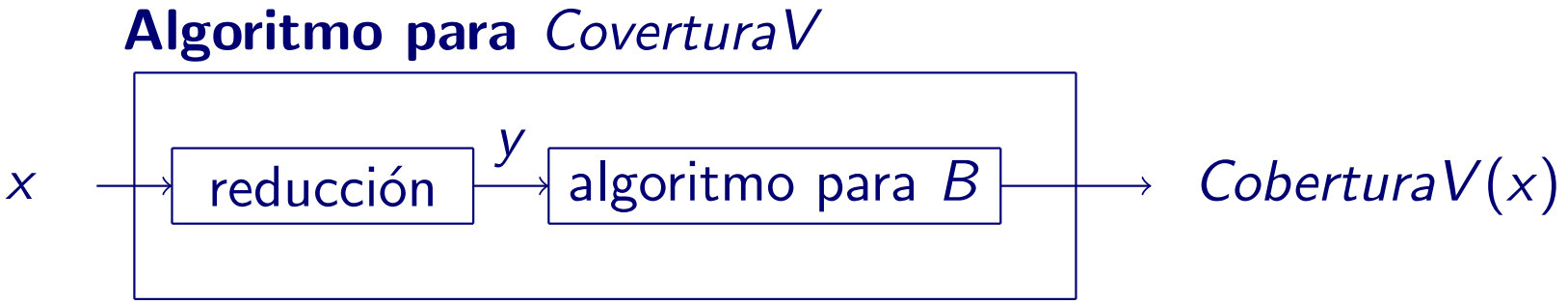
$$CicloHam \leq B$$



- Transformamos la entrada de *Ciclo Hamiltoniano* en entrada de B **en tiempo polinómico**
- La solución de B nos da la solución para *Ciclo Hamiltoniano*
- Como *Ciclo Hamiltoniano* es intratable entonces B es intratable

Reducción de *Cobertura de Vértices* a B

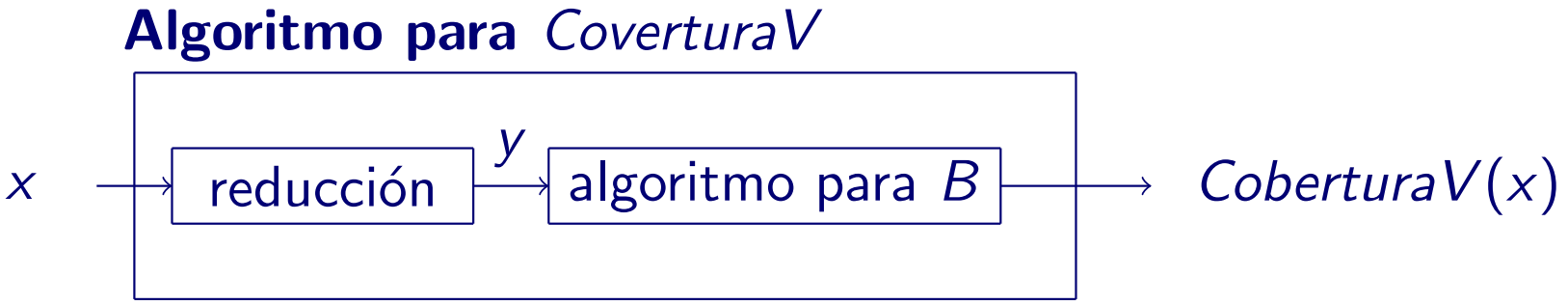
$$\mathit{CoberturaV} \leq B$$



- Transformamos la entrada de *Cobertura de Vértices* en entrada de B **en tiempo polinómico**
- La solución de B nos da la solución para *Cobertura de Vértices*

Reducción de *Cobertura de Vértices* a B

$$\mathit{CoberturaV} \leq B$$



- Transformamos la entrada de *Cobertura de Vértices* en entrada de B **en tiempo polinómico**
- La solución de B nos da la solución para *Cobertura de Vértices*
- Como *Cobertura de Vértices* es intratable entonces B es intratable

Las reducciones en tiempo polinómico son transitivas

- Si A es reducible a B y B es reducible a C entonces A es reducible a C
- Si A es intratable podemos demostrar que C es intratable demostrando primero que B es intratable (con $A \leq B$) y después que C es intratable (con $B \leq C$)

Veremos los siguientes ejemplos

- TSP es intratable
- Conjunto Independiente es intratable
- Problema general de planificación de películas es intratable
- Clique es intratable

TSP es intratable

- Empezamos con la definición de *TSP*

Problema: TSP
Entrada: n el número de ciudades, la matriz de distancias $n \times n$ y cota superior k.
Salida: ¿Existe un recorrido por las n ciudades, sin repeticiones y volviendo al punto de partida con distancia total $\leq k$?

TSP es intratable

- Vamos a utilizar una reducción de *Ciclo Hamiltoniano* a *TSP*, recordamos *Ciclo Hamiltoniano*

TSP es intratable

- Vamos a utilizar una reducción de *Ciclo Hamiltoniano* a *TSP*, recordamos *Ciclo Hamiltoniano*

Problema: Ciclo Hamiltoniano
Entrada: Un grafo G.
Salida: ¿Existe un ciclo hamiltoniano en G, es decir, un camino que visita una vez cada vértice y vuelve al vértice inicial?

TSP es intratable: reducir Ciclo Ham. a TSP

- Los dos problemas tratan de encontrar ciclos cortos, TSP en un grafo con pesos

TSP es intratable: reducir Ciclo Ham. a TSP

- Los dos problemas tratan de encontrar ciclos cortos, TSP en un grafo con pesos
- Para reducir *Ciclo Hamiltoniano* a *TSP* tenemos que convertir cada entrada de *Ciclo Hamiltoniano* (un grafo) en una entrada de *TSP* (ciudades y distancias)

TSP es intratable: reducir Ciclo Ham. a TSP

- Los dos problemas tratan de encontrar ciclos cortos, TSP en un grafo con pesos
- Para reducir *Ciclo Hamiltoniano* a *TSP* tenemos que convertir cada entrada de *Ciclo Hamiltoniano* (un grafo) en una entrada de *TSP* (ciudades y distancias)

CicloHamiltoniano(G)

```
1  n = número de vértices de G
2  for i = 1 to n
3       for j = 1 to n
4            if (i, j) es arista de G
5                 d(i, j) = 1
6            else d(i, j) = 2
7       Resultado TSP(n, d, n).
```

TSP es intratable: reducir Ciclo Ham. a TSP

- La reducción anterior tarda tiempo $O(n^2)$ (los dos for anidados), podemos resolver *Ciclo Hamiltoniano* en $O(n^2)+$ el tiempo de TSP

TSP es intratable: reducir Ciclo Ham. a TSP

- La reducción anterior tarda tiempo $O(n^2)$ (los dos for anidados), podemos resolver *Ciclo Hamiltoniano* en $O(n^2)+$ el tiempo de TSP
- La reducción funciona porque la respuesta de *Ciclo Hamiltoniano* con entrada G es la misma que *TSP* con entrada (n, d, n):

TSP es intratable: reducir Ciclo Ham. a TSP

- La reducción anterior tarda tiempo $O(n^2)$ (los dos for anidados), podemos resolver *Ciclo Hamiltoniano* en $O(n^2)+$ el tiempo de TSP
- La reducción funciona porque la respuesta de *Ciclo Hamiltoniano* con entrada G es la misma que *TSP* con entrada (n, d, n):
  - Si G tiene un circuito hamiltoniano $(v_1, \ldots, v_n)$ entonces el mismo recorrido para (n, d, n) tiene distancia total $\sum_{i=1}^{n-1} d(v_i, v_{i+1}) + d(v_n, v_1) = n$

TSP es intratable: reducir Ciclo Ham. a TSP

- La reducción anterior tarda tiempo $O(n^2)$ (los dos for anidados), podemos resolver *Ciclo Hamiltoniano* en $O(n^2)+$ el tiempo de TSP
- La reducción funciona porque la respuesta de *Ciclo Hamiltoniano* con entrada G es la misma que *TSP* con entrada (n, d, n):
  - Si G tiene un circuito hamiltoniano $(v_1, \ldots, v_n)$ entonces el mismo recorrido para (n, d, n) tiene distancia total $\sum_{i=1}^{n-1} d(v_i, v_{i+1}) + d(v_n, v_1) = n$
  - Si G no tiene un circuito hamiltoniano entonces (n, d, n) no tiene recorrido con distancia total $\leq n$ porque un recorrido así sólo puede pasar por n distancias 1, luego pasa por n aristas del grafo y sería un ciclo hamiltoniano de G.

TSP es intratable: reducir Ciclo Ham. a TSP

- La reducción anterior tarda tiempo $O(n^2)$ (los dos for anidados), podemos resolver *Ciclo Hamiltoniano* en $O(n^2)+$ el tiempo de TSP
- La reducción funciona porque la respuesta de *Ciclo Hamiltoniano* con entrada G es la misma que *TSP* con entrada (n, d, n):
  - Si G tiene un circuito hamiltoniano $(v_1, \ldots, v_n)$ entonces el mismo recorrido para (n, d, n) tiene distancia total $\sum_{i=1}^{n-1} d(v_i, v_{i+1}) + d(v_n, v_1) = n$
  - Si G no tiene un circuito hamiltoniano entonces (n, d, n) no tiene recorrido con distancia total $\leq n$ porque un recorrido así sólo puede pasar por n distancias 1, luego pasa por n aristas del grafo y sería un ciclo hamiltoniano de G.
- Tenemos una reducción eficiente de *Ciclo Hamiltoniano* a *TSP*. Como *Ciclo Hamiltoniano* es intratable entonces *TSP* **es intratable**.

Conjunto Independiente es intratable

- Empezamos con la definición de *Conjunto Independiente*
- Un conjunto de vértices S de G es *independiente* si no hay ninguna arista (i, j) de G con los dos vértices en S.

Conjunto Independiente es intratable

- Empezamos con la definición de *Conjunto Independiente*
- Un conjunto de vértices S de G es *independiente* si no hay ninguna arista (i, j) de G con los dos vértices en S.
- Cuanto más grande S más difícil es que sea independiente

Conjunto Independiente es intratable

- Empezamos con la definición de *Conjunto Independiente*
- Un conjunto de vértices S de G es *independiente* si no hay ninguna arista (i, j) de G con los dos vértices en S.
- Cuanto más grande S más difícil es que sea independiente

Problema: Conjunto Independiente
Entrada: Un grafo G (con vértices V) y $k \in \mathbb{N}$.
Salida: ¿Existe un conjunto independiente de k vértices de G?

Conjunto Independiente es intratable

- Empezamos con la definición de *Conjunto Independiente*
- Un conjunto de vértices S de G es *independiente* si no hay ninguna arista (i, j) de G con los dos vértices en S.
- Cuanto más grande S más difícil es que sea independiente

Problema: Conjunto Independiente
Entrada: Un grafo G (con vértices V) y $k \in \mathbb{N}$.
Salida: ¿Existe un conjunto independiente de k vértices de G?

- Vamos a utilizar una reducción de *Cobertura de Vértices* (VC) a *Conjunto Independiente*
- (recordad) Cuanto más pequeño k más difícil es la Cobertura de Vértices

Conjunto Independiente es intratable

- Empezamos con la definición de *Conjunto Independiente*
- Un conjunto de vértices S de G es *independiente* si no hay ninguna arista (i, j) de G con los dos vértices en S.
- Cuanto más grande S más difícil es que sea independiente

Problema: Conjunto Independiente
Entrada: Un grafo G (con vértices V) y $k \in \mathbb{N}$.
Salida: ¿Existe un conjunto independiente de k vértices de G?

- Vamos a utilizar una reducción de *Cobertura de Vértices* (VC) a *Conjunto Independiente*
- (recordad) Cuanto más pequeño k más difícil es la Cobertura de Vértices

Problema: Cobertura de Vértices
Entrada: Un grafo G (con vértices V) y $k \in \mathbb{N}$.
Salida: ¿Existe un conjunto U de k vértices de G tal que cada arista (i, j) de G cumple que $i \in U$ ó $j \in U$?

Conjunto Independiente es intratable: reducir Cobertura de Vértices a Conjunto Independiente

- Los dos problemas tratan de encontrar conjuntos de vértices, el primero con un vértice de cada arista, el segundo que no contenga aristas
- Si U es un conjunto independiente de G entonces $V - U$ es un cubrimiento de los vértices de G (ya que si $V - U$ no cubre una arista (i, j) de G entonces los dos vértices de esa arista están en U)

Conjunto Independiente es intratable: reducir Cobertura de Vértices a Conjunto Independiente

- Los dos problemas tratan de encontrar conjuntos de vértices, el primero con un vértice de cada arista, el segundo que no contenga aristas
- Si U es un conjunto independiente de G entonces $V - U$ es un cubrimiento de los vértices de G (ya que si $V - U$ no cubre una arista (i, j) de G entonces los dos vértices de esa arista están en U)
- Para reducir *Cobertura de Vértices* a *Conjunto Independiente* usamos la idea anterior

Conjunto Independiente es intratable: reducir Cobertura de Vértices a Conjunto Independiente

- Los dos problemas tratan de encontrar conjuntos de vértices, el primero con un vértice de cada arista, el segundo que no contenga aristas
- Si U es un conjunto independiente de G entonces $V - U$ es un cubrimiento de los vértices de G (ya que si $V - U$ no cubre una arista (i, j) de G entonces los dos vértices de esa arista están en U)
- Para reducir *Cobertura de Vértices* a *Conjunto Independiente* usamos la idea anterior

CoberturaVértices(G, k)

1 $k' = |V| - k$
2 Resultado ConjuntoIndependiente(G, k').

Conjunto Independiente es intratable: reducir Cobertura de Vértices a Conjunto Independiente

- La reducción anterior tarda tiempo $O(1)$, podemos resolver *Cobertura de Vértices* en el tiempo de *Conjunto Independiente*

Conjunto Independiente es intratable: reducir Cobertura de Vértices a Conjunto Independiente

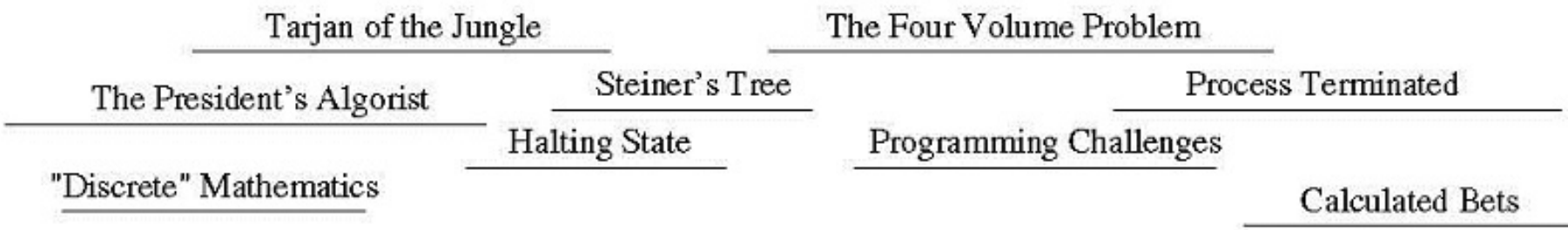
- La reducción anterior tarda tiempo $O(1)$, podemos resolver *Cobertura de Vértices* en el tiempo de *Conjunto Independiente*
- Es correcta, la respuesta de *Cobertura de Vértices* con entrada G, k es la misma que *Conjunto Independiente* con entrada $(G, |V| - k)$

Conjunto Independiente es intratable: reducir Cobertura de Vértices a Conjunto Independiente

- La reducción anterior tarda tiempo $O(1)$, podemos resolver *Cobertura de Vértices* en el tiempo de *Conjunto Independiente*
- Es correcta, la respuesta de *Cobertura de Vértices* con entrada G, k es la misma que *Conjunto Independiente* con entrada $(G, |V| - k)$
- Tenemos una reducción eficiente de *Cobertura de Vértices* a *Conjunto Independiente*. Como *Cobertura de Vértices* es intratable entonces *Conjunto Independiente* **es intratable**.

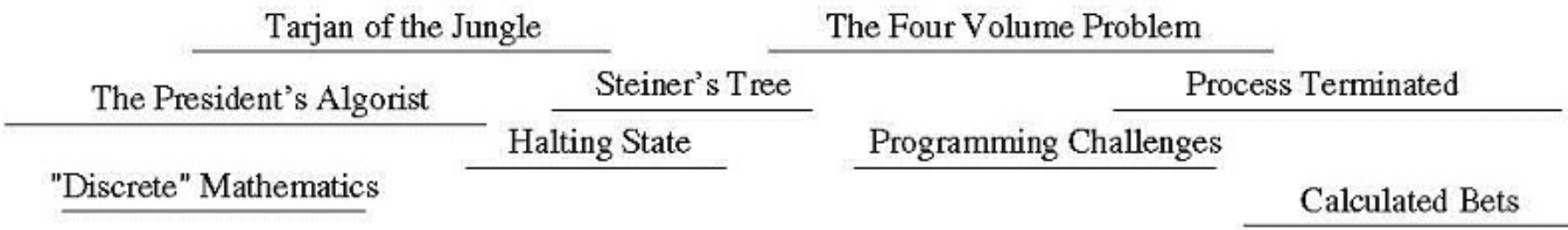
El problema de planificación de películas

- Imagínate que eres es un **actor muy cotizado**, que tienes ofertas para protagonizar n películas
- En cada oferta viene especificado en el primer y último día de rodaje
- Para aceptar el trabajo, debes comprometerte a estar disponible durante todo este período entero

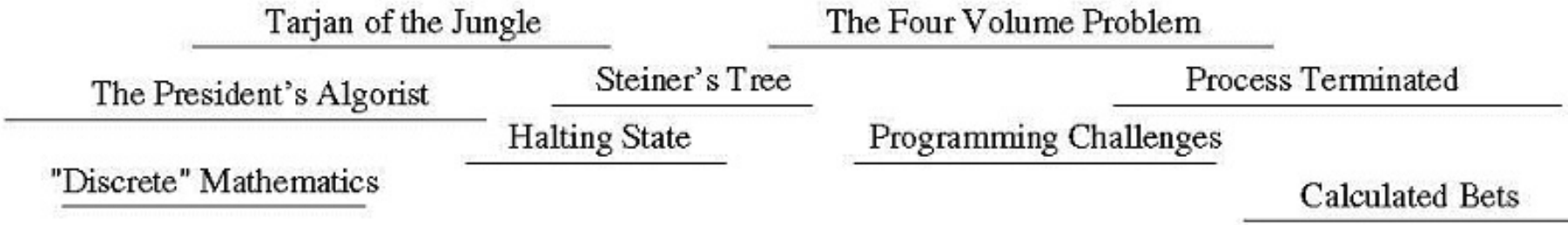


El problema de planificación de películas

- Imagínate que eres es un **actor muy cotizado**, que tienes ofertas para protagonizar n películas
- En cada oferta viene especificado en el primer y último día de rodaje
- Para aceptar el trabajo, debes comprometerte a estar disponible durante todo este período entero
- Por lo tanto no puedes aceptar simultáneamente dos papeles cuyos intervalos se superpongan



El problema de planificación de películas



- Para un artista como tú, los criterios para aceptar un trabajo son claros: quieres ganar tanto **dinero** como sea posible
- Debido a que cada una de estas películas paga la misma tarifa por película, esto implica que buscas la mayor cantidad posible de puestos de trabajo (intervalos) de tal manera que dos cualesquiera de ellos no estén en conflicto entre sí

El problema de planificación de películas

- Tú (o tu agente) debes resolver el siguiente problema de algoritmia:

  Problema: Planificación de películas
  Entrada: Un conjunto I de n intervalos, $k \in \mathbb{N}$
  Salida: ¿Cuál es el mayor conjunto de intervalos de I que dos a dos sean disjuntos? o ¿qué películas puedo hacer ganando la mayor cantidad de dinero?

El problema de planificación de películas

- Tú (o tu agente) debes resolver el siguiente problema de algoritmia:

  Problema: Planificación de películas
  Entrada: Un conjunto I de n intervalos, $k \in \mathbb{N}$
  Salida: ¿Cuál es el mayor conjunto de intervalos de I que dos a dos sean disjuntos? o ¿qué películas puedo hacer ganando la mayor cantidad de dinero?
- No es muy difícil encontrar un algoritmo eficiente (¿ideas?)

El problema de planificación de películas

- Tú (o tu agente) debes resolver el siguiente problema de algoritmia:

  Problema: Planificación de películas
  Entrada: Un conjunto I de n intervalos, $k \in \mathbb{N}$
  Salida: ¿Cuál es el mayor conjunto de intervalos de I que dos a dos sean disjuntos? o ¿qué películas puedo hacer ganando la mayor cantidad de dinero?
- No es muy difícil encontrar un algoritmo eficiente (¿ideas?)
- Pero vamos a considerar otro problema más difícil ...

El problema GENERAL de planificación de películas

- Más difícil: un proyecto de película no tiene porqué rodarse en un sólo intervalo, puede tener un calendario discontinuo

El problema GENERAL de planificación de películas

- Más difícil: un proyecto de película no tiene porqué rodarse en un sólo intervalo, puede tener un calendario discontinuo
- Por ejemplo "Tarzán de la jungla" se rodará en Enero-Marzo y Mayo-Junio, "Terminator" se rodará en Abril y en Agosto, "Babe" se rodará en Junio-Julio

El problema GENERAL de planificación de películas

- Más difícil: un proyecto de película no tiene porqué rodarse en un sólo intervalo, puede tener un calendario discontinuo
- Por ejemplo “Tarzán de la jungla” se rodará en Enero-Marzo y Mayo-Junio, “Terminator” se rodará en Abril y en Agosto, “Babe” se rodará en Junio-Julio
- El mismo actor puede rodar “Tarzán de la jungla” y “Terminator” pero no “Tarzán de la jungla” y “Babe”

El problema GENERAL de planificación de películas

- Más difícil: un proyecto de película no tiene porqué rodarse en un sólo intervalo, puede tener un calendario discontinuo
- Por ejemplo "Tarzán de la jungla" se rodará en Enero-Marzo y Mayo-Junio, "Terminator" se rodará en Abril y en Agosto, "Babe" se rodará en Junio-Julio
- El mismo actor puede rodar "Tarzán de la jungla" y "Terminator" pero no "Tarzán de la jungla" y "Babe"
- Vamos a ver que la versión decisional es **intratable**
  Problema: Planificación general de películas
  Entrada: Un conjunto I de n *conjuntos de intervalos*, $k \in \mathbb{N}$
  Salida: ¿Existen k *conjuntos de intervalos* de I que dos a dos sean disjuntos?

Prob. GENERAL de planif. de películas es intratable

- Necesitamos hacer una reducción desde otro problema que sepamos intratable

Prob. GENERAL de planif. de películas es intratable

- Necesitamos hacer una reducción desde otro problema que sepamos intratable
- Vamos a intentarlo desde *Conjunto Independiente*

Prob. GENERAL de planif. de películas es intratable

- Necesitamos hacer una reducción desde otro problema que sepamos intratable
- Vamos a intentarlo desde *Conjunto Independiente*
- ¿En qué se parecen los dos problemas? ...
  - Los dos pretenden seleccionar el subconjunto más grande posible – de vértices y de películas resp.

Prob. GENERAL de planif. de películas es intratable

- Necesitamos hacer una reducción desde otro problema que sepamos intratable
- Vamos a intentarlo desde *Conjunto Independiente*
- ¿En qué se parecen los dos problemas? ...
  - Los dos pretenden seleccionar el subconjunto más grande posible – de vértices y de películas resp.
  - Intentemos traducir los vértices en películas

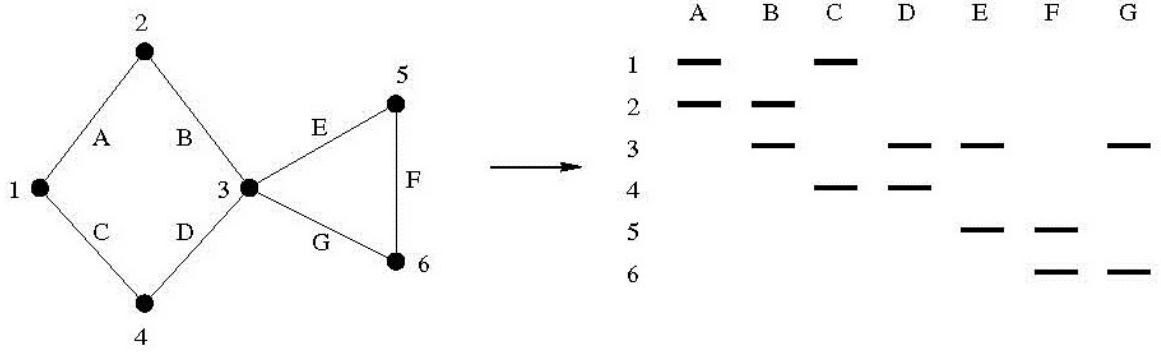
Prob. GENERAL de planif. de películas es intratable

- Necesitamos hacer una reducción desde otro problema que sepamos intratable
- Vamos a intentarlo desde *Conjunto Independiente*
- ¿En qué se parecen los dos problemas? ...
  - Los dos pretenden seleccionar el subconjunto más grande posible – de vértices y de películas resp.
  - Intentemos traducir los vértices en películas
  - Intentemos traducir vértices con una arista en películas incompatibles

Prob. GENERAL de planif. de películas es intratable

ConjuntoIndependiente(G, k)

1 m =número de aristas de G
2 n =número de vértices de G
3 $I = \emptyset$
4 **for** $j = 1$ **to** n
5 $\quad$ $pelicula(j) = \emptyset$
6 $\quad$ $I = anadir(I, pelicula(j))$
7 **for** la $i$ésima arista de G (x, y), $1 \leq i \leq m$
8 $\quad$ $pelicula(x) = anadir(pelicula(x), [i, i + 0{,}5])$
9 $\quad$ $pelicula(y) = anadir(pelicula(y), [i, i + 0{,}5])$
10 Resultado GralPaInifPeliculas(I, k).



Prob. GENERAL de planif. de películas es intratable

Prob. GENERAL de planif. de películas es intratable

- Cada arista se traduce en un intervalo

Prob. GENERAL de planif. de películas es intratable

- Cada arista se traduce en un intervalo
- Cada vértice es una película que contiene los intervalos de las aristas desde ese vértice

Prob. GENERAL de planif. de películas es intratable

- Cada arista se traduce en un intervalo
- Cada vértice es una película que contiene los intervalos de las aristas desde ese vértice
- Dos vértices unidos por una arista (prohibido en el *Conjunto Independiente*) definen un par de películas que comparten un intervalo (prohibido en el calendario del actor) y viceversa

Prob. GENERAL de planif. de películas es intratable

- Cada arista se traduce en un intervalo
- Cada vértice es una película que contiene los intervalos de las aristas desde ese vértice
- Dos vértices unidos por una arista (prohibido en el *Conjunto Independiente*) definen un par de películas que comparten un intervalo (prohibido en el calendario del actor) y viceversa
- Los mayores subconjuntos que satisfacen los dos problemas son los mismos

Prob. GENERAL de planif. de películas es intratable

- Cada arista se traduce en un intervalo
- Cada vértice es una película que contiene los intervalos de las aristas desde ese vértice
- Dos vértices unidos por una arista (prohibido en el *Conjunto Independiente*) definen un par de películas que comparten un intervalo (prohibido en el calendario del actor) y viceversa
- Los mayores subconjuntos que satisfacen los dos problemas son los mismos
- Tenemos una reducción, un algoritmo eficiente para el *problema gral. de planificación de películas* nos da un algoritmo eficiente para *Conjunto Independiente*

Prob. GENERAL de planif. de películas es intratable

- Cada arista se traduce en un intervalo
- Cada vértice es una película que contiene los intervalos de las aristas desde ese vértice
- Dos vértices unidos por una arista (prohibido en el *Conjunto Independiente*) definen un par de películas que comparten un intervalo (prohibido en el calendario del actor) y viceversa
- Los mayores subconjuntos que satisfacen los dos problemas son los mismos
- Tenemos una reducción, un algoritmo eficiente para el *problema gral. de planificación de películas* nos da un algoritmo eficiente para *Conjunto Independiente*
- Como *Conjunto Independiente* es intratable, *problema gral. de planificación de películas* es intratable

Clique es intratable

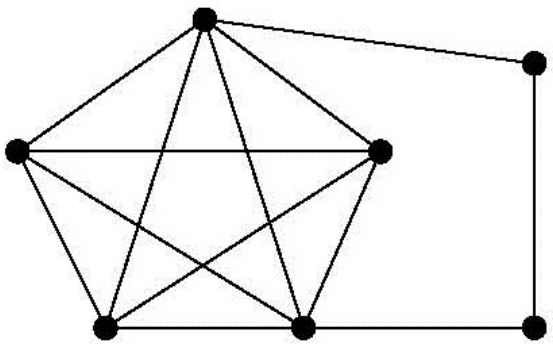
- Empezamos con la definición de Clique

Clique es intratable

- Empezamos con la definición de Clique
- Un **clique social** es un conjunto de personas que todos conocen a todos. Un **clique** en un grafo es un conjunto de vértices que todos están unidos a todos

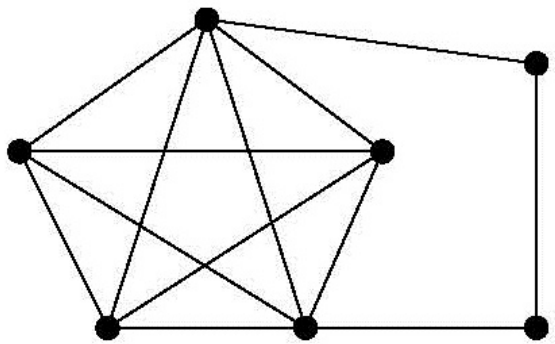
Clique es intratable

- Empezamos con la definición de Clique
- Un **clique social** es un conjunto de personas que todos conocen a todos. Un **clique** en un grafo es un conjunto de vértices que todos están unidos a todos



Clique es intratable

- Empezamos con la definición de Clique
- Un **clique social** es un conjunto de personas que todos conocen a todos. Un **clique** en un grafo es un conjunto de vértices que todos están unidos a todos



Problema: Clique
Entrada: Un grafo G (con vértices V) y $k \in \mathbb{N}$.
Salida: ¿Tiene G un clique de k vértices, es decir, existe un conjunto U de k vértices de G tal que cada par $x, y \in U$, existe la arista (x, y) en G?

Clique es intratable

- Cuanto más grande k más difícil es Clique

Clique es intratable

- Cuanto más grande k más difícil es Clique
- En un grafo social los cliques corresponden a lugares de trabajo, vecindarios, parroquias, escuelas ...

Clique es intratable

- Cuanto más grande k más difícil es Clique
- En un grafo social los cliques corresponden a lugares de trabajo, vecindarios, parroquias, escuelas ...
- ¿Y en el grafo de internet?

Clique es intratable: reducir Conj. Independiente a Clique

- Vamos a utilizar una reducción de *Conjunto Independiente* a *Clique*
- Los dos problemas tratan de encontrar conjuntos de vértices, el primero que no contenga aristas, el segundo que contenga todas las aristas

Clique es intratable: reducir Conj. Independiente a Clique

- Vamos a utilizar una reducción de *Conjunto Independiente* a *Clique*
- Los dos problemas tratan de encontrar conjuntos de vértices, el primero que no contenga aristas, el segundo que contenga todas las aristas
- ¿Cómo podemos relacionarlos?
- Para reducir *Conjunto Independiente* a *Clique* cambiamos aristas por no aristas y no aristas por aristas, es decir, **complementamos el grafo**

Clique es intratable: reducir Conj. Independiente a Clique

- Vamos a utilizar una reducción de *Conjunto Independiente* a *Clique*
- Los dos problemas tratan de encontrar conjuntos de vértices, el primero que no contenga aristas, el segundo que contenga todas las aristas
- ¿Cómo podemos relacionarlos?
- Para reducir *Conjunto Independiente* a *Clique* cambiamos aristas por no aristas y no aristas por aristas, es decir, **complementamos el grafo**

ConjuntoIndependiente(G, k)

1 Construir un grafo G' con los mismos vértices de G
2 aristas de $G' = \emptyset$
3 **for** $x \in V$
4 **for** $y \in V$
5 Si (x, y) no es arista de G la añadimos a las aristas de
6 Resultado Clique(G', k).

Clique es intratable: reducir Conj. Independiente a Clique

- La reducción anterior tarda tiempo $O(n^2)$, podemos resolver *Conjunto Independiente* en el tiempo de *Clique* $+O(n^2)$

Clique es intratable: reducir Conj. Independiente a Clique

- La reducción anterior tarda tiempo $O(n^2)$, podemos resolver *Conjunto Independiente* en el tiempo de *Clique* $+O(n^2)$
- Tenemos una reducción eficiente de *Conjunto Independiente* a *Clique*. Como *Conjunto Independiente* es intratable entonces *Clique* **es intratable**

Clique es intratable: reducir Conj. Independiente a Clique

- La reducción anterior tarda tiempo $O(n^2)$, podemos resolver *Conjunto Independiente* en el tiempo de *Clique* $+O(n^2)$
- Tenemos una reducción eficiente de *Conjunto Independiente* a *Clique*. Como *Conjunto Independiente* es intratable entonces *Clique* **es intratable**
- Hemos hecho una cadena transitiva:

$$\text{Cobertura de Vértices} \leq \text{Conjunto Independiente}$$
$$\text{Conjunto Independiente} \leq \text{Clique}$$

Clique es intratable: reducir Conj. Independiente a Clique

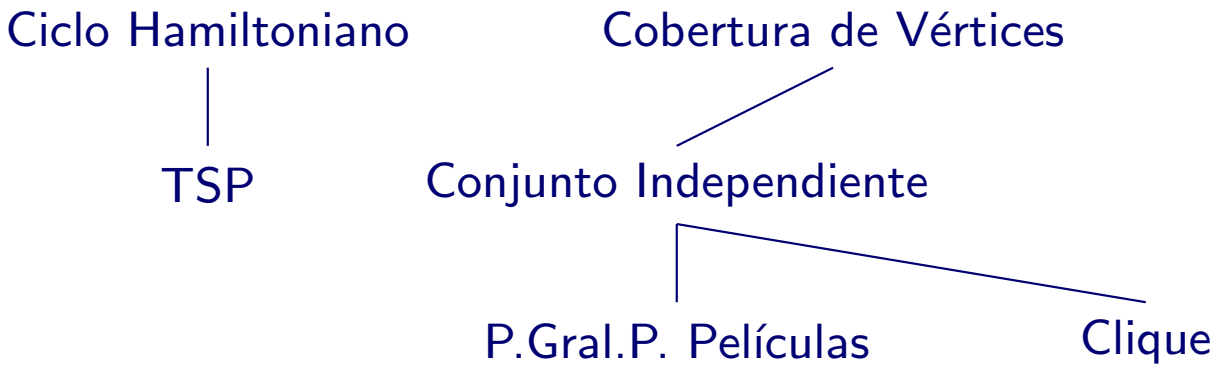
- La reducción anterior tarda tiempo $O(n^2)$, podemos resolver *Conjunto Independiente* en el tiempo de *Clique* $+O(n^2)$
- Tenemos una reducción eficiente de *Conjunto Independiente* a *Clique*. Como *Conjunto Independiente* es intratable entonces *Clique* **es intratable**
- Hemos hecho una cadena transitiva:

$$\text{Cobertura de Vértices} \leq \text{Conjunto Independiente}$$
$$\text{Conjunto Independiente} \leq \text{Clique}$$

- Nota: La misma reducción (complementar el grafo) sirve para reducir *Clique* a *Conjunto Independiente*, luego los dos problemas son equivalentes

Resumen

- Hemos visto que los siguientes 4 problemas son intratables: *TSP*, *Conjunto Independiente*, *Problema GENERAL de planificación de películas*, *Clique*
- Para ello hemos usado (sin justificar) que *Ciclo Hamiltoniano* y *Cobertura de Vértices* son intratables

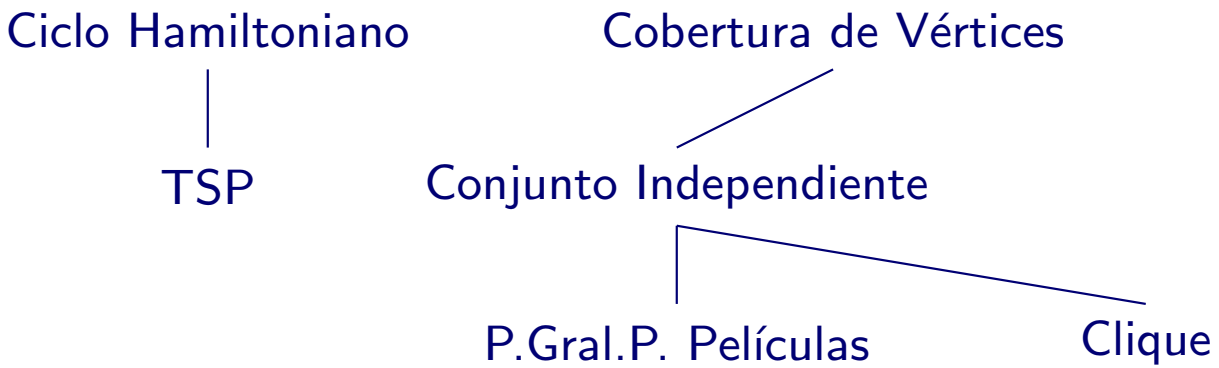


- Hemos visto sólo **reducciones sencillas:**
  - Equivalencia: CoverturaVért es lo mismo que ConjuntoIndep. ConjuntoIndep es lo mismo que Clique
  - Restricción: CicloHam es un caso particular de TSP
  - Sustitución local/Diseño de componentes: Traducir las aristas de ConjIndependiente a intervalos de Películas

Contenido de este tema

Problemas vistos

- Hemos visto que los siguientes 4 problemas son intratables: *TSP*, *Conjunto Independiente*, *Problema GENERAL de planificación de películas*, *Clique*
- Para ello hemos usado (sin justificar) que *Ciclo Hamiltoniano* y *Cobertura de Vértices* son intratables



- ¿Por qué son intratables *Ciclo Hamiltoniano* y *Cobertura de Vértices*?

SAT

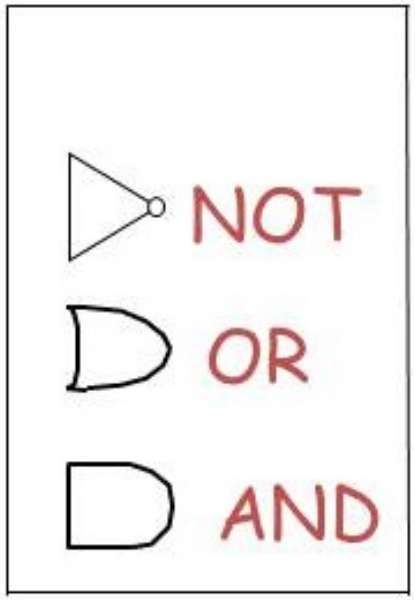
- Para demostrar la intratabilidad de otros problemas debemos empezar con un único problema que sea **EL** candidato a intratable
- La **madre de todos los intratables** en un problema de lógica llamado Satisfacibilidad (SAT)

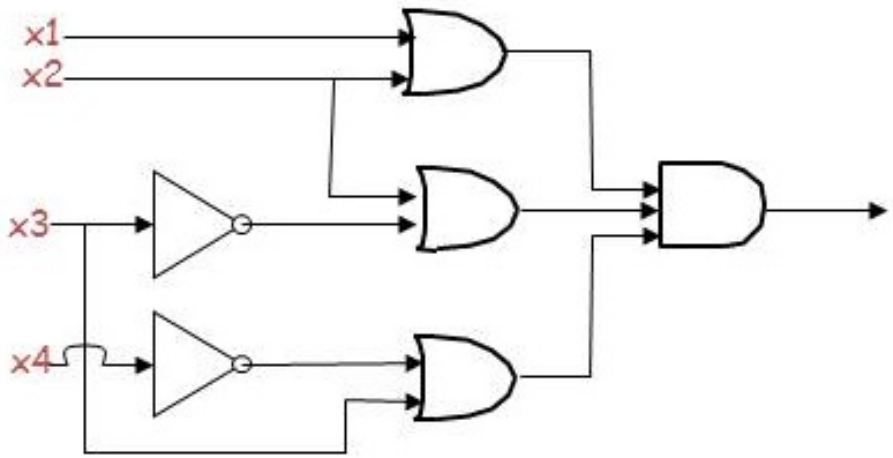
SAT

Problema: SAT
Entrada: Un circuito booleano en CNF C con una única salida.
Salida: ¿Existe una asignación de las entradas de C que da salida Cierto?

SAT

Problema: SAT
Entrada: Un circuito booleano en CNF C con una única salida.
Salida: ¿Existe una asignación de las entradas de C que da salida Cierto?





SAT: ejemplos

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$$

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1)$$

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1) \wedge (\neg x_3)$$

SAT

- Existen múltiples concursos para programas que resuelvan SAT (SAT solvers)
- Es el paradigma de problema intratable, es ampliamente aceptado que es intratable (aunque no se ha demostrado)
- Daremos alguna razón que hace sospechar que SAT es intratable

Variantes de SAT

- Si restringimos a una sola entrada por puerta OR es fácil encontrar un algoritmo eficiente

Variantes de SAT

- Si restringimos a una sola entrada por puerta OR es fácil encontrar un algoritmo eficiente
- Si restringimos a dos entradas por puerta OR se puede encontrar un algoritmo eficiente, este problema se llama 2-SAT

Variantes de SAT

- Si restringimos a una sola entrada por puerta OR es fácil encontrar un algoritmo eficiente
- Si restringimos a dos entradas por puerta OR se puede encontrar un algoritmo eficiente, este problema se llama 2-SAT
- ¿Y si restringimos a tres entradas por puerta OR?

Variantes de SAT

- Si restringimos a una sola entrada por puerta OR es fácil encontrar un algoritmo eficiente
- Si restringimos a dos entradas por puerta OR se puede encontrar un algoritmo eficiente, este problema se llama 2-SAT
- ¿Y si restringimos a tres entradas por puerta OR?
- Vamos a ver que este último caso es intratable y nos es útil para demostrar que otros problemas son intratables

3-SAT

Problema: 3-SAT
Entrada: Un circuito booleano en CNF C en el que **cada puerta OR contiene exactamente 3 entradas distintas**, con una única salida.
Salida: ¿Existe una asignación de las entradas de C que da salida Cierto?

3-SAT

Problema: 3-SAT
Entrada: Un circuito booleano en CNF C en el que **cada puerta OR contiene exactamente 3 entradas distintas**, con una única salida.
Salida: ¿Existe una asignación de las entradas de C que da salida Cierto?

- Ejemplo de entrada:
  $(x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge (x_1 \vee x_2 \vee x_3)$

3-SAT

Problema: 3-SAT
Entrada: Un circuito booleano en CNF C en el que **cada puerta OR contiene exactamente 3 entradas distintas**, con una única salida.
Salida: ¿Existe una asignación de las entradas de C que da salida Cierto?

- Ejemplo de entrada:
  $(x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge (x_1 \vee x_2 \vee x_3)$
- Podemos demostrar que 3-SAT es intratable haciendo una reducción de SAT a 3-SAT

Reducción de SAT a 3-SAT

¿Cómo transformamos un circuito de SAT en uno de 3-SAT?

- Si tenemos un OR con una sola entrada x creamos dos variables nuevas v_1, v_2 y sustituimos el OR por

$$(x \vee v_1 \vee v_2) \wedge (x \vee \neg v_1 \vee v_2) \wedge (x \vee v_1 \vee \neg v_2) \wedge (x \vee \neg v_1 \vee \neg v_2)$$

Reducción de SAT a 3-SAT

¿Cómo transformamos un circuito de SAT en uno de 3-SAT?

- Si tenemos un OR con una sola entrada x creamos dos variables nuevas v_1, v_2 y sustituimos el OR por

  $$(x \vee v_1 \vee v_2) \wedge (x \vee \neg v_1 \vee v_2) \wedge (x \vee v_1 \vee \neg v_2) \wedge (x \vee \neg v_1 \vee \neg v_2)$$

- Si tenemos un OR con dos entradas x, y creamos una variable nueva v y sustituimos el OR por

  $$(x \vee y \vee v) \wedge (x \vee y \vee \neg v)$$

Reducción de SAT a 3-SAT

¿Cómo transformamos un circuito de SAT en uno de 3-SAT?

- Si tenemos un OR con una sola entrada x creamos dos variables nuevas v_1, v_2 y sustituimos el OR por

$$(x \vee v_1 \vee v_2) \wedge (x \vee \neg v_1 \vee v_2) \wedge (x \vee v_1 \vee \neg v_2) \wedge (x \vee \neg v_1 \vee \neg v_2)$$

- Si tenemos un OR con dos entradas x, y creamos una variable nueva v y sustituimos el OR por

$$(x \vee y \vee v) \wedge (x \vee y \vee \neg v)$$

- Si tenemos un OR con **más de tres** entradas $y_1, \ldots, y_n$
  - Creamos $n-3$ variables nuevas $v_1, \ldots, v_{n-3}$
  - Sustituimos $(y_1 \vee \ldots \vee y_n)$ por $F_1 \wedge \ldots \wedge F_{n-2}$ con

$$F_1 = (y_1 \vee y_2 \vee \neg v_1)$$

$$F_j = (v_{j-1} \vee y_{j+1} \vee \neg v_j),\ 2 \leq j \leq n-3$$

$$F_{n-2} = (v_{n-3} \vee y_{n-1} \vee y_n)$$

Reducción de SAT a 3-SAT

- Si tenemos un OR con **más de tres** entradas $y_1, \ldots, y_n$
  - Creamos $n-3$ variables nuevas $v_1, \ldots, v_{n-3}$
  - Sustituimos $(y_1 \vee \ldots \vee y_n)$ por $F_1 \wedge \ldots \wedge F_{n-2}$ con

$$F_1 = (y_1 \vee y_2 \vee \neg v_1)$$

$$F_j = (v_{j-1} \vee y_{j+1} \vee \neg v_j),\ 2 \leq j \leq n-3$$

$$F_{n-2} = (v_{n-3} \vee y_{n-1} \vee y_n)$$

Reducción de SAT a 3-SAT

- Si tenemos un OR con **más de tres** entradas $y_1, \ldots, y_n$
  - Creamos $n-3$ variables nuevas $v_1, \ldots, v_{n-3}$
  - Sustituimos $(y_1 \vee \ldots \vee y_n)$ por $F_1 \wedge \ldots \wedge F_{n-2}$ con

$$F_1 = (y_1 \vee y_2 \vee \neg v_1)$$

$$F_j = (v_{j-1} \vee y_{j+1} \vee \neg v_j),\ 2 \leq j \leq n-3$$

$$F_{n-2} = (v_{n-3} \vee y_{n-1} \vee y_n)$$

- El caso más complicado es este último ...
  - Si hay una asignación que hace cierta $(y_1 \vee \ldots \vee y_n)$ entonces hace cierto por lo menos un y_i, podemos hacer cierto $F_1 \wedge \ldots \wedge F_{n-2}$ con $v_1 = v_2 = \ldots = v_{i-2} =$falso y $v_{i-1} = v_i = \ldots = v_{n-3} =$cierto

Reducción de SAT a 3-SAT

- Si tenemos un OR con **más de tres** entradas $y_1, \ldots, y_n$
  - Creamos $n-3$ variables nuevas $v_1, \ldots, v_{n-3}$
  - Sustituimos $(y_1 \vee \ldots \vee y_n)$ por $F_1 \wedge \ldots \wedge F_{n-2}$ con

$$F_1 = (y_1 \vee y_2 \vee \neg v_1)$$

$$F_j = (v_{j-1} \vee y_{j+1} \vee \neg v_j),\ 2 \leq j \leq n-3$$

$$F_{n-2} = (v_{n-3} \vee y_{n-1} \vee y_n)$$

- El caso más complicado es este último ...
  - Si hay una asignación que hace cierta $(y_1 \vee \ldots \vee y_n)$ entonces hace cierto por lo menos un y_i, podemos hacer cierto $F_1 \wedge \ldots \wedge F_{n-2}$ con $v_1 = v_2 = \ldots = v_{i-2}$ =falso y $v_{i-1} = v_i = \ldots = v_{n-3}$ =cierto
  - Probemos todos los casos, si una asignación hace cierta y_1 ó y_2

Reducción de SAT a 3-SAT

- Si tenemos un OR con **más de tres** entradas $y_1, \ldots, y_n$
  - Creamos $n-3$ variables nuevas $v_1, \ldots, v_{n-3}$
  - Sustituimos $(y_1 \vee \ldots \vee y_n)$ por $F_1 \wedge \ldots \wedge F_{n-2}$ con

$$F_1 = (y_1 \vee y_2 \vee \neg v_1)$$

$$F_j = (v_{j-1} \vee y_{j+1} \vee \neg v_j),\ 2 \leq j \leq n-3$$

$$F_{n-2} = (v_{n-3} \vee y_{n-1} \vee y_n)$$

- El caso más complicado es este último ...
  - Si hay una asignación que hace cierta $(y_1 \vee \ldots \vee y_n)$ entonces hace cierto por lo menos un y_i, podemos hacer cierto $F_1 \wedge \ldots \wedge F_{n-2}$ con $v_1 = v_2 = \ldots = v_{i-2}$ =falso y $v_{i-1} = v_i = \ldots = v_{n-3}$ =cierto
  - Probemos todos los casos, si una asignación hace cierta y_1 ó y_2
  - si hace cierta y_i con $3 \leq i \leq n-2$

Reducción de SAT a 3-SAT

- Si tenemos un OR con **más de tres** entradas $y_1, \ldots, y_n$
  - Creamos $n-3$ variables nuevas $v_1, \ldots, v_{n-3}$
  - Sustituimos $(y_1 \vee \ldots \vee y_n)$ por $F_1 \wedge \ldots \wedge F_{n-2}$ con

$$F_1 = (y_1 \vee y_2 \vee \neg v_1)$$

$$F_j = (v_{j-1} \vee y_{j+1} \vee \neg v_j),\ 2 \leq j \leq n-3$$

$$F_{n-2} = (v_{n-3} \vee y_{n-1} \vee y_n)$$

- El caso más complicado es este último ...
  - Si hay una asignación que hace cierta $(y_1 \vee \ldots \vee y_n)$ entonces hace cierto por lo menos un y_i, podemos hacer cierto $F_1 \wedge \ldots \wedge F_{n-2}$ con $v_1 = v_2 = \ldots = v_{i-2}$ =falso y $v_{i-1} = v_i = \ldots = v_{n-3}$ =cierto
  - Probemos todos los casos, si una asignación hace cierta y_1 ó y_2
  - si hace cierta y_i con $3 \leq i \leq n-2$
  - si hace cierta y_{n-1} ó y_n ...

Reducción de SAT a 3-SAT

- Si tenemos un OR con **más de tres** entradas $y_1, \ldots, y_n$
  - Creamos $n-3$ variables nuevas $v_1, \ldots, v_{n-3}$
  - Sustituimos $(y_1 \vee \ldots \vee y_n)$ por $F_1 \wedge \ldots \wedge F_{n-2}$ con

$$F_1 = (y_1 \vee y_2 \vee \neg v_1)$$

$$F_j = (v_{j-1} \vee y_{j+1} \vee \neg v_j),\ 2 \leq j \leq n-3$$

$$F_{n-2} = (v_{n-3} \vee y_{n-1} \vee y_n)$$

- El caso más complicado es este último ...
  - Si hay una asignación que hace cierta $(y_1 \vee \ldots \vee y_n)$ entonces hace cierto por lo menos un y_i, podemos hacer cierto $F_1 \wedge \ldots \wedge F_{n-2}$ con $v_1 = v_2 = \ldots = v_{i-2}$ =falso y $v_{i-1} = v_i = \ldots = v_{n-3}$ =cierto
  - Probemos todos los casos, si una asignación hace cierta y_1 ó y_2
  - si hace cierta y_i con $3 \leq i \leq n-2$
  - si hace cierta y_{n-1} ó y_n ...
- También es cierto que si una asignación hace cierta $F_1 \wedge \ldots \wedge F_{n-2}$ entonces la misma asignación restringida a las variables originales hace cierta $(y_1 \vee \ldots \vee y_n)$ ¿Por qué?

Reducción de SAT a 3-SAT

- Si tenemos un OR con **más de tres** entradas $y_1, \ldots, y_n$
  - Creamos $n-3$ variables nuevas $v_1, \ldots, v_{n-3}$
  - Sustituimos $(y_1 \vee \ldots \vee y_n)$ por $F_1 \wedge \ldots \wedge F_{n-2}$ con

$$F_1 = (y_1 \vee y_2 \vee \neg v_1)$$

$$F_j = (v_{j-1} \vee y_{j+1} \vee \neg v_j),\ 2 \leq j \leq n-3$$

$$F_{n-2} = (v_{n-3} \vee y_{n-1} \vee y_n)$$

- El caso más complicado es este último ...
  - Si hay una asignación que hace cierta $(y_1 \vee \ldots \vee y_n)$ entonces hace cierto por lo menos un y_i, podemos hacer cierto $F_1 \wedge \ldots \wedge F_{n-2}$ con $v_1 = v_2 = \ldots = v_{i-2}$ =falso y $v_{i-1} = v_i = \ldots = v_{n-3}$ =cierto
  - Probemos todos los casos, si una asignación hace cierta y_1 ó y_2
  - si hace cierta y_i con $3 \leq i \leq n-2$
  - si hace cierta y_{n-1} ó y_n ...
- También es cierto que si una asignación hace cierta $F_1 \wedge \ldots \wedge F_{n-2}$ entonces la misma asignación restringida a las variables originales hace cierta $(y_1 \vee \ldots \vee y_n)$ ¿Por qué?
- Así que tenemos una reducción de SAT a 3-SAT

Reducción de SAT a 3-SAT

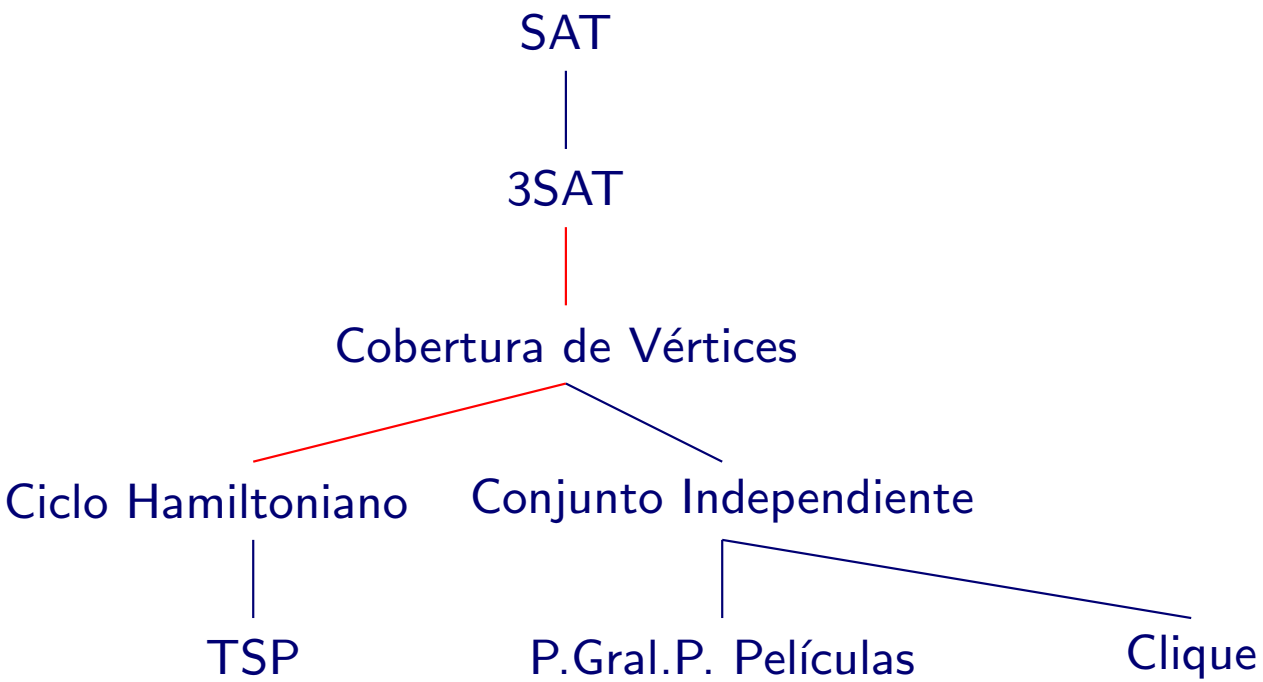
- ¿Cuánto tiempo tarda la reducción?

Reducción de SAT a 3-SAT

- ¿Cuánto tiempo tarda la reducción? $O(|C|)$, donde $|C|$ es el tamaño del circuito C

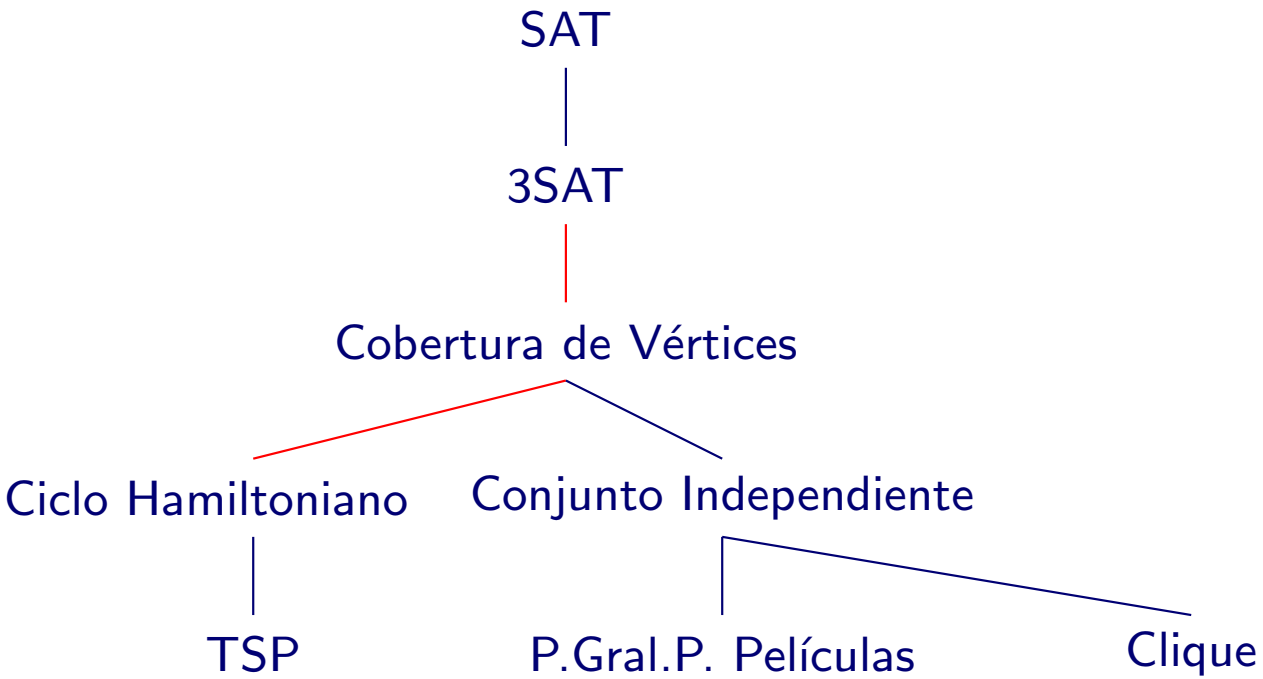
Problemas intratables vistos

- Hemos visto las reducciones que aparecen en negro



Problemas intratables vistos

- Hemos visto las reducciones que aparecen en negro
- Las que aparecen en rojo son más complicadas y no las veremos (algunas están en las secciones 9.5-9.8 del Skiena)



El arte de demostrar intratabilidad

- En general es más fácil pensar una reducción (o una demostración de intratabilidad) que explicarla/entenderla

El arte de demostrar intratabilidad

- En general es más fácil pensar una reducción (o una demostración de intratabilidad) que explicarla/entenderla
- Una sutil diferencia puede convertir un problema intratable en tratable o viceversa

El arte de demostrar intratabilidad

- En general es más fácil pensar una reducción (o una demostración de intratabilidad) que explicarla/entenderla
- Una sutil diferencia puede convertir un problema intratable en tratable o viceversa Camino más corto/camino más largo. Pasar por todos los vértices/aristas una sola vez

El arte de demostrar intratabilidad

- En general es más fácil pensar una reducción (o una demostración de intratabilidad) que explicarla/entenderla
- Una sutil diferencia puede convertir un problema intratable en tratable o viceversa Camino más corto/camino más largo. Pasar por todos los vértices/aristas una sola vez
- Lo primero que hay que hacer si sospechamos que un problema es intratable es mirar el libro de Garey Johnson

El arte de demostrar intratabilidad

- En general es más fácil pensar una reducción (o una demostración de intratabilidad) que explicarla/entenderla
- Una sutil diferencia puede convertir un problema intratable en tratable o viceversa Camino más corto/camino más largo. Pasar por todos los vértices/aristas una sola vez
- Lo primero que hay que hacer si sospechamos que un problema es intratable es mirar el libro de Garey Johnson

- Si esto no funciona, para reducir un intratable A a un problema nuevo N

El arte de demostrar intratabilidad

- En general es más fácil pensar una reducción (o una demostración de intratabilidad) que explicarla/entenderla
- Una sutil diferencia puede convertir un problema intratable en tratable o viceversa Camino más corto/camino más largo. Pasar por todos los vértices/aristas una sola vez
- Lo primero que hay que hacer si sospechamos que un problema es intratable es mirar el libro de Garey Johnson

- Si esto no funciona, para reducir un intratable A a un problema nuevo N
  - Haz A lo más simple posible
  - Haz N lo más difícil posible
  - Seleccionar el A adecuado por las razones adecuadas
  - Si te quedas atascado, alterna entre intentar ver que N es intratable y encontrar un algoritmo eficiente para N

Usando intratabilidad para diseñar algoritmos

Usando intratabilidad para diseñar algoritmos

- La teoría de la intratabilidad es muy útil para diseñar algoritmos, aunque sólo dé resultados negativos.

Usando intratabilidad para diseñar algoritmos

- La teoría de la intratabilidad es muy útil para diseñar algoritmos, aunque sólo dé resultados negativos.
- Permite al diseñador centrar sus esfuerzos más productivamente, no darse de cabezazos contra la pared. Al menos divide los esfuerzos entre las dos posibilidades: algoritmo eficiente o comparar con intratables

Usando intratabilidad para diseñar algoritmos

- La teoría de la intratabilidad es muy útil para diseñar algoritmos, aunque sólo dé resultados negativos.
- Permite al diseñador centrar sus esfuerzos más productivamente, no darse de cabezazos contra la pared. Al menos divide los esfuerzos entre las dos posibilidades: algoritmo eficiente o comparar con intratables
- La teoría de la intratabilidad nos permite identificar qué propiedades hacen un problema difícil.

Usando intratabilidad para diseñar algoritmos

- La teoría de la intratabilidad es muy útil para diseñar algoritmos, aunque sólo dé resultados negativos.
- Permite al diseñador centrar sus esfuerzos más productivamente, no darse de cabezazos contra la pared. Al menos divide los esfuerzos entre las dos posibilidades: algoritmo eficiente o comparar con intratables
- La teoría de la intratabilidad nos permite identificar qué propiedades hacen un problema difícil. Tener una intuición de qué problemas van a ser intratables es importante para un diseñador, y sólo se consigue con experiencia demostrando intratabilidad.

Recomendable

- Leer las secciones 9.5-9.8 del Skiena
- Incluyen demostraciones de intratabilidad más complicadas y en general filosofía de estas demostraciones

Contenido de este tema

Problemas débilmente intratables

- En el contexto de datos masivos, tiempo por encima de cuasilineal es demasiado

Problemas débilmente intratables

- En el contexto de datos masivos, tiempo por encima de cuasilineal es demasiado
- Vamos a presentar los **intratables débiles** que son problemas que no sabemos resolver en tiempo esencialmente menor que cuadrático

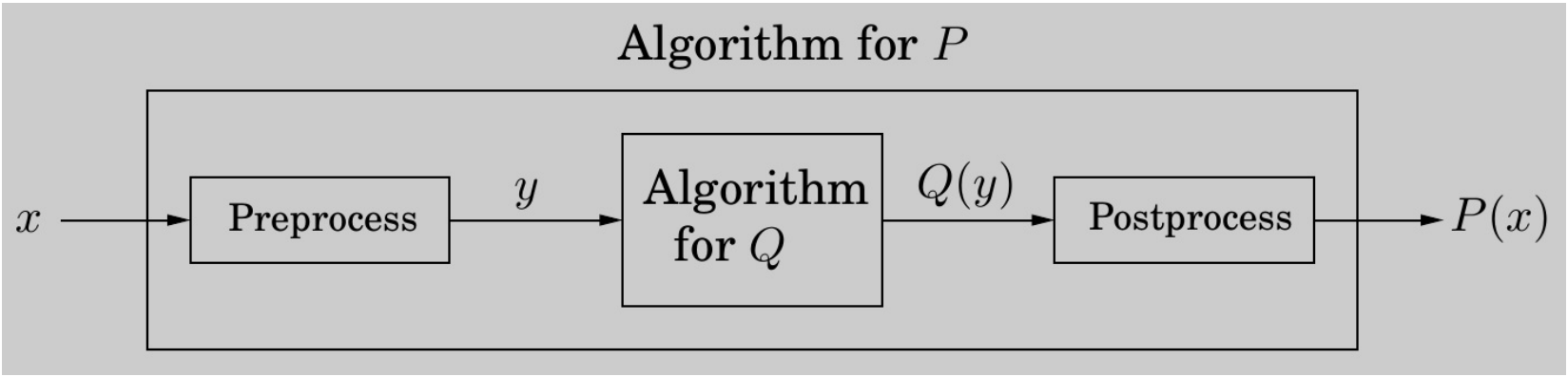
Problemas débilmente intratables

- En el contexto de datos masivos, tiempo por encima de cuasilineal es demasiado
- Vamos a presentar los **intratables débiles** que son problemas que no sabemos resolver en tiempo esencialmente menor que cuadrático
- Necesitamos usar reducciones muy rápidas para la intratabilidad débil

Problemas débilmente intratables

- En el contexto de datos masivos, tiempo por encima de cuasilineal es demasiado
- Vamos a presentar los **intratables débiles** que son problemas que no sabemos resolver en tiempo esencialmente menor que cuadrático
- Necesitamos usar reducciones muy rápidas para la intratabilidad débil
- Al conseguir compararlos unos con otros los hacemos **algorítmicamente equivalentes**: un algoritmo cuasilineal para uno de ellos nos daría uno para todos

Reducciones



$\mathrm{A}(x)$

1 **for** $i = 1$ **to** k
2 Calcula y_i (a partir de $z_1, \ldots, z_{i-1}$)
3 Calcula $z_i = B(y_i)$
4 Cálculos finales a partir de $z_1, \ldots, z_k$
5 Resultado $\mathrm{A}(x)$

¿Qué cota de tiempo podemos poner para comparar candidatos a intratables débiles?

Alinear 2 secuencias

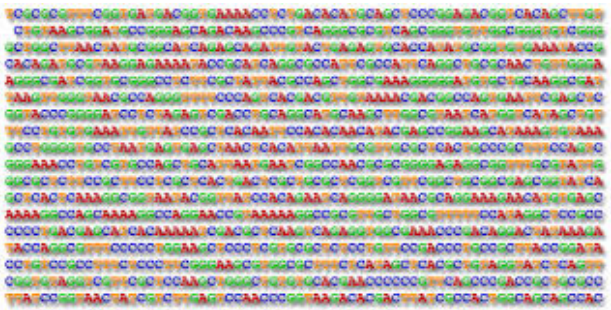
- Los humanos tenemos alrededor de 3 Gigas de ADN en cada célula (una palabra de $3 * 10^9$ letras)

Alinear 2 secuencias

- Los humanos tenemos alrededor de 3 Gigas de ADN en cada célula (una palabra de $3 * 10^9$ letras)
- Como va cortado en cromosomas vamos a pensar que nos conformamos con trabajar con uno de ellos, 1 mega (palabra de 10^6 letras)

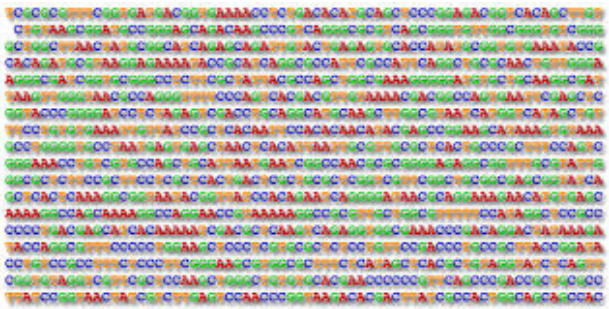
Alinear 2 secuencias

- Los humanos tenemos alrededor de 3 Gigas de ADN en cada célula (una palabra de $3 * 10^9$ letras)
- Como va cortado en cromosomas vamos a pensar que nos conformamos con trabajar con uno de ellos, 1 mega (palabra de 10^6 letras)
- Vamos a pensar en el problema de **comparar** el mismo cromosoma de 2 individuos



Alinear 2 secuencias

- Los humanos tenemos alrededor de 3 Gigas de ADN en cada célula (una palabra de $3 * 10^9$ letras)
- Como va cortado en cromosomas vamos a pensar que nos conformamos con trabajar con uno de ellos, 1 mega (palabra de 10^6 letras)
- Vamos a pensar en el problema de **comparar** el mismo cromosoma de 2 individuos



Supongamos que queremos comparar nuestro fragmento de un mega con todos los de una base de datos como Genbank (1.600 millones de secuencias)

Alineamiento

Datos de entrada: Dos palabras o cadenas
Buscamos: Una optimización de las inserciones, borrados y cambios que me han llevado de una a otra

```
GAATTCAGTTA
GGATCAGTA
```

Alineamiento

Datos de entrada: Dos palabras o cadenas
Buscamos: Una optimización de las inserciones, borrados y cambios que me han llevado de una a otra

```
GAATTCAGTTA
GGATCAGTA
```

Alineamiento

Datos de entrada: Dos palabras o cadenas
Buscamos: Una optimización de las inserciones, borrados y cambios que me han llevado de una a otra

```
GAATTCAGTTA
GGATCAGTA
```


```
GAATTCAGTTA
GGA-TCAGTA-
```

Coste de un alineamiento

- Cada inserción tiene un coste (por ejemplo 3)
- Cada sustitución tiene un coste (5)
- Objetivo: encontrar el alineamiento que minimiza el coste total

```
GAATTCAGTTA
GGA-TCAGTA-
```

Alineamiento

- Los mejores algoritmos que se conocen para resolver **realmente** este problema tardan tiempo $n^2/\log n$

Alineamiento

- Los mejores algoritmos que se conocen para resolver **realmente** este problema tardan tiempo $n^2/\log n$
- para nuestro cromosoma de $n = 10^6$ estamos hablando de 10^{12} instrucciones (1 minuto en un solo procesador)

Alineamiento

- Los mejores algoritmos que se conocen para resolver **realmente** este problema tardan tiempo $n^2/\log n$
- para nuestro cromosoma de $n = 10^6$ estamos hablando de 10^{12} instrucciones (1 minuto en un solo procesador)
- Lo queremos hacer para un número enorme de casos (1.600 millones)

¿Por qué no bajamos de tiempo cuadrático?

- Compararlo con otros problemas que tampoco sepamos mejorar de tiempo cuadrático nos puede ayudar a entender las razones
- Aquí **la madre de los intratables débiles** es el problema de **Vectores Ortogonales (OV)**

Vectores ortogonales

Problema: OV

Datos de entrada: Dos conjuntos S, T de n cadenas binarias cada uno de longitud $d \geq \log n$ ($S, T \subseteq \{0,1\}^d$)

Buscamos: ¿Existen $a \in S$ y $b \in T$ que sean ortogonales ($\sum_i a_i b_i = 0$)?

Vectores ortogonales

Problema: OV

Datos de entrada: Dos conjuntos S, T de n cadenas binarias cada uno de longitud $d \geq \log n$ ($S, T \subseteq \{0,1\}^d$)

Buscamos: ¿Existen $a \in S$ y $b \in T$ que sean ortogonales ($\sum_i a_i b_i = 0$)?

- El mejor algoritmo conocido es esencialmente probar todos con todos (tiempo cuadrático)

Alineamiento es tan difícil como Vectores ortogonales

$\mathrm{OV}(S, T, n, d)$

1 $x =$ CONVERSION1(S)
2 $y =$ CONVERSION2(T)
3 **if** Alineamiento$(x, y) \leq 5d + X$ Resultado True
4 **if** Alineamiento$(x, y) \geq 15d + X$ Resultado False

Alineamiento es tan difícil como Vectores ortogonales

$\mathrm{OV}(S, T, n, d)$

1 $x =$ CONVERSION1(S)
2 $y =$ CONVERSION2(T)
3 **if** Alineamiento$(x, y) \leq 5d + X$ Resultado True
4 **if** Alineamiento$(x, y) \geq 15d + X$ Resultado False

- CONVERSION1 cambia cada 0 por 0111 y cada 1 por 0001

Alineamiento es tan difícil como Vectores ortogonales

$\mathrm{OV}(S, T, n, d)$

1 $x =$ CONVERSION1(S)
2 $y =$ CONVERSION2(T)
3 **if** Alineamiento$(x, y) \leq 5d + X$ Resultado True
4 **if** Alineamiento$(x, y) \geq 15d + X$ Resultado False

- CONVERSION1 cambia cada 0 por 0111 y cada 1 por 0001
- CONVERSION2 cambia cada 0 por 0011 y cada 1 por 1111

Alineamiento es tan difícil como Vectores ortogonales

$\mathrm{OV}(S, T, n, d)$

1 $x =$ CONVERSION1(S)
2 $y =$ CONVERSION2(T)
3 **if** Alineamiento$(x, y) \leq 5d + X$ Resultado True
4 **if** Alineamiento$(x, y) \geq 15d + X$ Resultado False

- CONVERSION1 cambia cada 0 por 0111 y cada 1 por 0001
- CONVERSION2 cambia cada 0 por 0011 y cada 1 por 1111
- Es más complicado (bloques entre uno y otro, etc)

Alineamiento es tan difícil como Vectores ortogonales

$\mathrm{OV}(S, T, n, d)$

1 $x =$ CONVERSION1(S)
2 $y =$ CONVERSION2(T)
3 **if** Alineamiento$(x, y) \leq 5d + X$ Resultado True
4 **if** Alineamiento$(x, y) \geq 15d + X$ Resultado False

- CONVERSION1 cambia cada 0 por 0111 y cada 1 por 0001
- CONVERSION2 cambia cada 0 por 0011 y cada 1 por 1111
- Es más complicado (bloques entre uno y otro, etc)
- **Si tuviera un algoritmo con coste $n^{2-\epsilon}$ para Alineamiento me saldría tiempo $O(n^{2-\epsilon})$ para OV**

Otros problemas similares

- Largest Common Subsequence (LCS)

Otros problemas similares

- Largest Common Subsequence (LCS)
- **Colinearidad**: Dados n puntos en el plano, comprobar que no hay 3 alineados

Otros problemas similares

- Largest Common Subsequence (LCS)
- **Colinearidad**: Dados n puntos en el plano, comprobar que no hay 3 alineados
- **Suma de 3**: Dados n números enteros entre $-n^4$ y n^4, encontrar 3 de ellos x, y, z que cumplan $x + y = z$

Otros problemas similares

- Largest Common Subsequence (LCS)
- **Colinearidad**: Dados n puntos en el plano, comprobar que no hay 3 alineados
- **Suma de 3**: Dados n números enteros entre $-n^4$ y n^4, encontrar 3 de ellos x, y, z que cumplan $x + y = z$
- Cálculo del diámetro de un grafo de n vértices y $O(n)$ aristas

Otros problemas similares

- Largest Common Subsequence (LCS)
- **Colinearidad**: Dados n puntos en el plano, comprobar que no hay 3 alineados
- **Suma de 3**: Dados n números enteros entre $-n^4$ y n^4, encontrar 3 de ellos x, y, z que cumplan $x + y = z$
- Cálculo del diámetro de un grafo de n vértices y $O(n)$ aristas
- Les pasa lo mismo que al de antes, sólo los sabemos resolver en tiempo cuadrático

Algunos punteros

- Tenemos así una nueva familia de intratables débiles qué son problemas que no sabemos resolver en tiempo subcuadrático
- Son intratables cuando queremos resolver casos masivos

Algunos punteros

- Tenemos así una nueva familia de intratables débiles qué son problemas que no sabemos resolver en tiempo subcuadrático
- Son intratables cuando queremos resolver casos masivos
- De todo esto se ocupa un reciente campo de investigación que es **la complejidad de granularidad fina**

Algunos punteros

- Tenemos así una nueva familia de intratables débiles qué son problemas que no sabemos resolver en tiempo subcuadrático
- Son intratables cuando queremos resolver casos masivos
- De todo esto se ocupa un reciente campo de investigación que es **la complejidad de granularidad fina**
- Si queréis saber algo más de ello podéis echar un vistazo a la página de la profesora Virginia Williams del MIT `https://people.csail.mit.edu/virgi/`

Resumen

- Queríamos conocer los límites de la computación y así llegamos al concepto de intratable
- Pero ha sido esa misma computación la que nos ha dado suficientes datos como para tener que replantearnos esa intratabilidad, necesitamos algoritmos realmente rápidos para poder trabajar con todos los datos que hemos conseguido
- Y una gran fuente de datos es la genómica ... y estamos en medio de una pandemia mundial con un virus cuyas mutaciones nos interesa mucho controlar ...

Contenido de este tema

Definición de NP-completo y NP-difícil

Definición

A es **NP-difícil** si SAT es reducible a A (SAT $\leq A$).

Definición de NP-completo y NP-difícil

Definición

A es **NP-difícil** si SAT es reducible a A (SAT $\leq A$). También llamado problema difícil, intratable, NP-hard

Definición de NP-completo y NP-difícil

Definición

A es **NP-difícil** si SAT es reducible a A (SAT $\leq A$). También llamado problema difícil, intratable, NP-hard

Definición

A es **NP-completo** si SAT es reducible a A y A es reducible a SAT (SAT $\leq A$ y $A \leq$ SAT), es decir, si A es equivalente a SAT

Definición de NP-completo y NP-difícil

Definición

A es **NP-difícil** si SAT es reducible a A (SAT $\leq A$). También llamado problema difícil, intratable, NP-hard

Definición

A es **NP-completo** si SAT es reducible a A y A es reducible a SAT (SAT $\leq A$ y $A \leq$ SAT), es decir, si A es equivalente a SAT

- **Nota:** Los NP-difíciles vistos hasta ahora (3SAT, Cobertura de Vértices, Ciclo Hamiltoniano, TSP, Conjunto Independiente, Clique) son todos NP-completos, es decir, equivalentes a SAT

Definición de NP-completo y NP-difícil

Definición

A es **NP-difícil** si SAT es reducible a A (SAT $\leq A$). También llamado problema difícil, intratable, NP-hard

Definición

A es **NP-completo** si SAT es reducible a A y A es reducible a SAT (SAT $\leq A$ y $A \leq$ SAT), es decir, si A es equivalente a SAT

- **Nota:** Los NP-difíciles vistos hasta ahora (3SAT, Cobertura de Vértices, Ciclo Hamiltoniano, TSP, Conjunto Independiente, Clique) son todos NP-completos, es decir, equivalentes a SAT
- Seguramente algunos NP-difíciles no son NP-completos (y son más difíciles que SAT), por ejemplo el juego del ajedrez

La teoría de los NP-completos

- En 1982 Stephen Cook ganó el premio Turing de la ACM por sus contribuciones a esta teoría.
- Para miles de problemas fundamentales de optimización, inteligencia artificial, combinatoria, lógica, etc, **la pregunta abierta de si son computacionalmente intratables** ha sido y es muy difícil de responder: No conocemos algoritmos eficientes y no podemos probar que no existan.

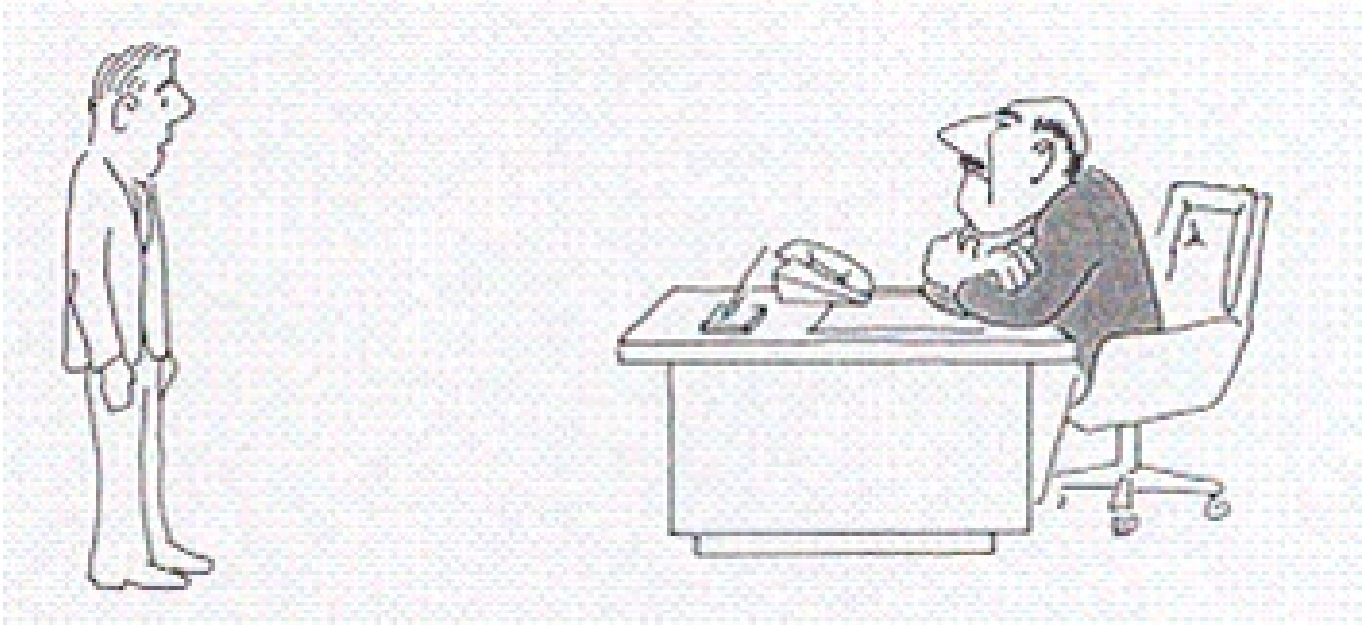
La teoría de los NP-completos

- En 1982 Stephen Cook ganó el premio Turing de la ACM por sus contribuciones a esta teoría.
- Para miles de problemas fundamentales de optimización, inteligencia artificial, combinatoria, lógica, etc, **la pregunta abierta de si son computacionalmente intratables** ha sido y es muy difícil de responder: No conocemos algoritmos eficientes y no podemos probar que no existan.
- El progreso que supone la teoría de los NP-completos es que demuestra que **todos estos problemas son equivalentes** en el sentido de que un algoritmo eficiente para uno de ellos supondría un algoritmo eficiente para cada uno de ellos.
- Todas esas preguntas abiertas son en realidad **una sola pregunta** debido a la equivalencia.

La teoría de los NP-completos

- En 1982 Stephen Cook ganó el premio Turing de la ACM por sus contribuciones a esta teoría.
- Para miles de problemas fundamentales de optimización, inteligencia artificial, combinatoria, lógica, etc, **la pregunta abierta de si son computacionalmente intratables** ha sido y es muy difícil de responder: No conocemos algoritmos eficientes y no podemos probar que no existan.
- El progreso que supone la teoría de los NP-completos es que demuestra que **todos estos problemas son equivalentes** en el sentido de que un algoritmo eficiente para uno de ellos supondría un algoritmo eficiente para cada uno de ellos.
- Todas esas preguntas abiertas son en realidad **una sola pregunta** debido a la equivalencia.
- Existe un premio de un millón de dólares para el que consiga resolverla

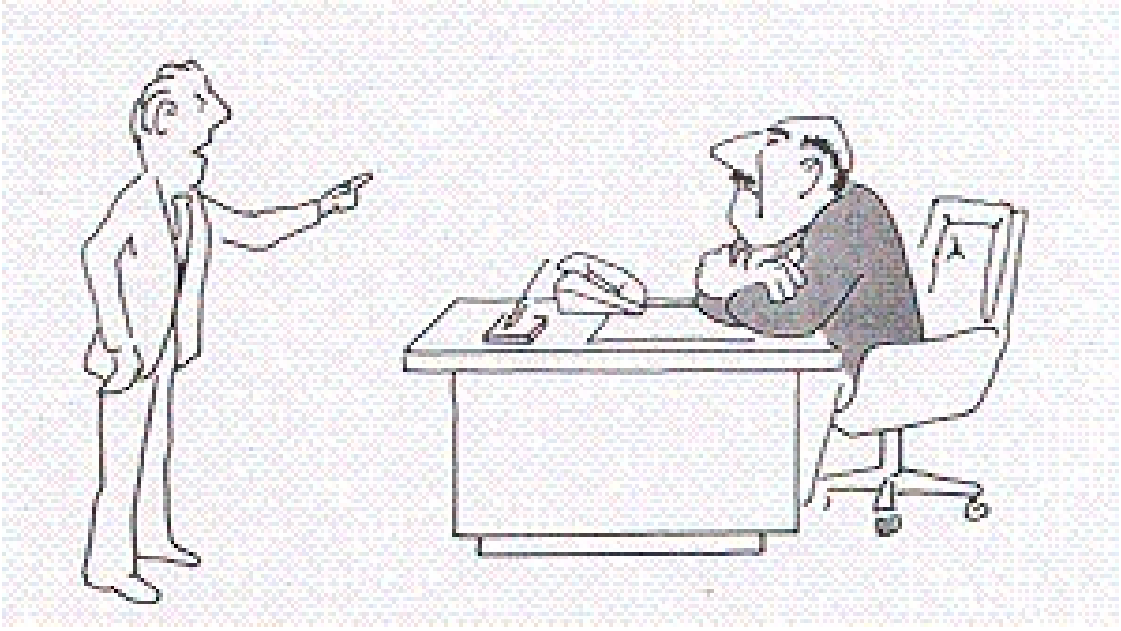
Una aplicación práctica



Te encuentras en una situación muy embarazosa:

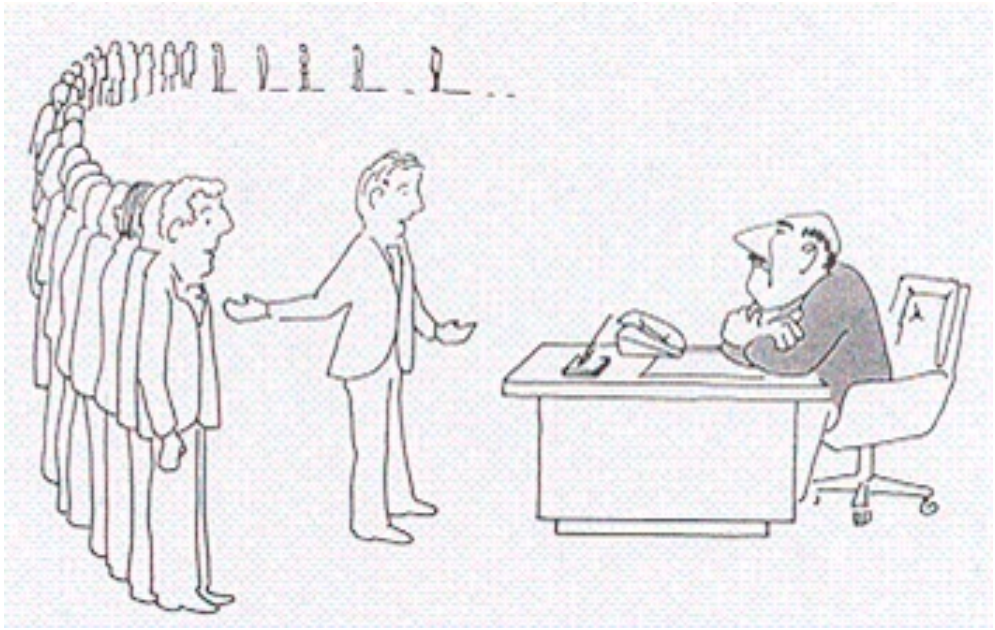
"No puedo encontrar un algoritmo eficiente, me temo que no estoy a la altura"

Te gustaría poder decir ...



"No puedo encontrar un algoritmo eficiente porque no existe"

Usando la teoría de los NP-completos ...



"No puedo encontrar un algoritmo eficiente pero tampoco pueden ninguno de estos informáticos famosos"

Intro. del libro sobre NP-completos "Garey, Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman. 1978."

¿Y ahora qué?

- ¿Qué hacemos con los problemas intratables?

¿Y ahora qué?

- ¿Qué hacemos con los problemas intratables?
- Si hemos demostrado que es intratable, es porque para empezar necesitábamos resolverlo

¿Y ahora qué?

- ¿Qué hacemos con los problemas intratables?
- Si hemos demostrado que es intratable, es porque para empezar necesitábamos resolverlo
- No nos conformamos con saber que no hay algoritmos eficientes que los resuelvan completamente, nos sigue interesando resolverlos de alguna manera

¿Y ahora qué?

- ¿Qué hacemos con los problemas intratables?
- Si hemos demostrado que es intratable, es porque para empezar necesitábamos resolverlo
- No nos conformamos con saber que no hay algoritmos eficientes que los resuelvan completamente, nos sigue interesando resolverlos de alguna manera
- En este curso veremos varias aproximaciones

¿Y ahora qué?

- ¿Qué hacemos con los problemas intratables?
- Si hemos demostrado que es intratable, es porque para empezar necesitábamos resolverlo
- No nos conformamos con saber que no hay algoritmos eficientes que los resuelvan completamente, nos sigue interesando resolverlos de alguna manera
- En este curso veremos varias aproximaciones
  1. **Algoritmos aproximados**: en algunos casos podemos encontrar eficientemente respuestas *cercanas* a la óptima

¿Y ahora qué?

- ¿Qué hacemos con los problemas intratables?
- Si hemos demostrado que es intratable, es porque para empezar necesitábamos resolverlo
- No nos conformamos con saber que no hay algoritmos eficientes que los resuelvan completamente, nos sigue interesando resolverlos de alguna manera
- En este curso veremos varias aproximaciones
  1. **Algoritmos aproximados**: en algunos casos podemos encontrar eficientemente respuestas *cercanas* a la óptima
  2. **Algoritmos probabilistas**: En algunos casos utilizar el azar nos da algoritmos eficientes

¿Y ahora qué?

- ¿Qué hacemos con los problemas intratables?
- Si hemos demostrado que es intratable, es porque para empezar necesitábamos resolverlo
- No nos conformamos con saber que no hay algoritmos eficientes que los resuelvan completamente, nos sigue interesando resolverlos de alguna manera
- En este curso veremos varias aproximaciones
  1. **Algoritmos aproximados**: en algunos casos podemos encontrar eficientemente respuestas *cercanas* a la óptima
  2. **Algoritmos probabilistas**: En algunos casos utilizar el azar nos da algoritmos eficientes
  3. **Problemas con datos masivos (big data)**:

¿Y ahora qué?

- ¿Qué hacemos con los problemas intratables?
- Si hemos demostrado que es intratable, es porque para empezar necesitábamos resolverlo
- No nos conformamos con saber que no hay algoritmos eficientes que los resuelvan completamente, nos sigue interesando resolverlos de alguna manera
- En este curso veremos varias aproximaciones
  1. **Algoritmos aproximados**: en algunos casos podemos encontrar eficientemente respuestas *cercanas* a la óptima
  2. **Algoritmos probabilistas**: En algunos casos utilizar el azar nos da algoritmos eficientes
  3. **Problemas con datos masivos (big data)**:
     - Estructuras de datos especializadas
     - Compresión