Complejidad Computacional

Algoritmos

Depto. de Computación, Facultade de Informática, Universidade da Coruña

Carlos Gómez Rodríguez
carlos.gomez@udc.es

Outline

1. Introducción. $\mathcal{P}$ y $\mathcal{NP}$.
2. Máquinas de Turing
3. $\mathcal{NP}$-completitud

Outline

1. Introducción. $\mathcal{P}$ y $\mathcal{NP}$.
2. Máquinas de Turing
3. $\mathcal{NP}$-completitud

Algoritmia y complejidad computacional

- Algoritmia: diseño de algoritmos *específicos*, lo más eficientes posibles, para un problema dado
  - Ejemplo: la *ordenación por selección* es $O(n^2)$, la *ordenación por montículos* es $O(nlog(n))$.
- Complejidad computacional: estudio del *problema*, considerando globalmente todos los algoritmos posibles para resolverlo (¡incluso los que no conocemos!)
  - Ejemplo: ¿cuál es la mejor complejidad temporal que podemos conseguir para ordenar un array de tamaño n?
  - Sabemos que podemos hacerlo en $O(nlog(n))$; pero ¿podemos hacerlo más rápido?
  - Si la respuesta es "sí", podemos demostrarlo encontrando un algoritmo más rápido. Pero si es "no", buscar algoritmos mejores sería perder el tiempo. Necesitamos razonar sobre el problema en su conjunto, más allá de algoritmos concretos $\rightarrow$ complejidad computacional.

Problemas tratables e intratables (1)

- Nos centraremos en un aspecto dado: qué problemas son *tratables* en la práctica.
- Se considera que un problema es tratable si se puede resolver en tiempo polinómico $O(n^k)$ para algún k. El conjunto de estos problemas se llama clase de complejidad $\mathcal{P}$.
  - Por ejemplo, sabemos que multiplicar dos números, ordenar una lista de valores, buscar el camino mínimo entre dos ciudades en un mapa, están en $\mathcal{P}$.

Problemas tratables e intratables (2)

Para otros problemas, no conocemos algoritmos en tiempo polinómico. Ejemplo: problema de la suma de subconjuntos.

- Dado un conjunto de n enteros (por ejemplo, $\{4, -2, 6, 3, 1, -28, 10, -17\}$). ¿Existe un subconjunto cuya suma sea 0?
- En este caso, sí: $4 + 3 + 10 + (-17) = 0$.

Problemas tratables e intratables (2)

Para otros problemas, no conocemos algoritmos en tiempo polinómico. Ejemplo: problema de la suma de subconjuntos.

- Dado un conjunto de n enteros (por ejemplo, $\{4, -2, 6, 3, 1, -28, 10, -17\}$). ¿Existe un subconjunto cuya suma sea 0?
- En este caso, sí: $4 + 3 + 10 + (-17) = 0$.

```
función Suma_Subconjuntos ( {i_1, ..., i_n} ) : bool
  visto := false
  para cada subconjunto {x_1, ..., x_k} ⊆ {i_1, ..., i_n}
    si x_1 + x_2 + ... + x_k = 0 entonces visto := true
  fin para;
  devolver visto
fin función
```

Problemas tratables e intratables (2)

Para otros problemas, no conocemos algoritmos en tiempo polinómico. Ejemplo: problema de la suma de subconjuntos.

- Dado un conjunto de n enteros (por ejemplo, $\{4, -2, 6, 3, 1, -28, 10, -17\}$). ¿Existe un subconjunto cuya suma sea 0?
- En este caso, sí: $4 + 3 + 10 + (-17) = 0$.

```
función Suma_Subconjuntos ( {i_1, ..., i_n} ) : bool
  visto := false
  para cada subconjunto {x_1, ..., x_k} ⊆ {i_1, ..., i_n}
    si x_1 + x_2 + ... + x_k = 0 entonces visto := true {O(n)}
  fin para;
  devolver visto
fin función
```

Problemas tratables e intratables (2)

Para otros problemas, no conocemos algoritmos en tiempo polinómico. Ejemplo: problema de la suma de subconjuntos.

- Dado un conjunto de n enteros (por ejemplo, $\{4, -2, 6, 3, 1, -28, 10, -17\}$). ¿Existe un subconjunto cuya suma sea 0?
- En este caso, sí: $4 + 3 + 10 + (-17) = 0$.

```
función Suma_Subconjuntos ( {i_1, ..., i_n} ) : bool
  visto := false
  para cada subconjunto {x_1, ..., x_k} ⊆ {i_1, ..., i_n}  {O(n2^n)}!
    si x_1 + x_2 + ... + x_k = 0 entonces visto := true  {O(n)}
  fin para;
  devolver visto
fin función
```

Problemas tratables e intratables (3)

- Para este problema no se conoce algoritmo polinómico $\rightarrow$ no sabemos si está en $\mathcal{P}$.
  - En teoría podría estar: si existe algoritmo polinómico pero no se ha descubierto (improbable).
- Este problema pertenece a una clase de complejidad llamada $\mathcal{NP}$
  - Ojo: nada que ver con "no polinómico", sino con "non-deterministic polynomial time" (veremos qué significa)

El problema de $\mathcal{P}$ y $\mathcal{NP}$ (1)

- Informalmente, $\mathcal{NP}$ es el conjunto de problemas cuya solución se puede comprobar en tiempo polinómico.
  - $\mathcal{P} \subseteq NP$: si tenemos un algoritmo polinómico para averiguar la solución, podemos usarlo para comprobarla (ej. suma de la subsecuencia máxima).
  - Pero hay problemas cuya solución es fácil de comprobar ($\mathcal{NP}$) pero difícil de obtener.
    - Suma de subconjuntos: podemos comprobar si un subconjunto es solución sumando sus elementos ($O(n)$) pero encontrar nosotros la solución es otra historia...
  - La "pregunta del millón" (literalmente) es si todos estos problemas están también en $\mathcal{P}$ o no, es decir, si $\mathcal{P} = \mathcal{NP}$ o no.

El problema de $\mathcal{P}$ y $\mathcal{NP}$ (2)

- Se cree que $\mathcal{P} \neq \mathcal{NP}$; pero hasta ahora no se ha logrado demostrarlo.
- Si efectivamente $\mathcal{P} \neq \mathcal{NP}$, nunca encontraremos soluciones polinómicas al problema de las sumas de subconjuntos (y otros problemas prácticos que están en $\mathcal{NP}$), porque sabremos que no existen (veremos por qué).
- Si $\mathcal{P} = \mathcal{NP}$, existirían soluciones polinómicas al problema de las sumas de subconjuntos y todos los demás problemas de $\mathcal{NP}$.

Outline

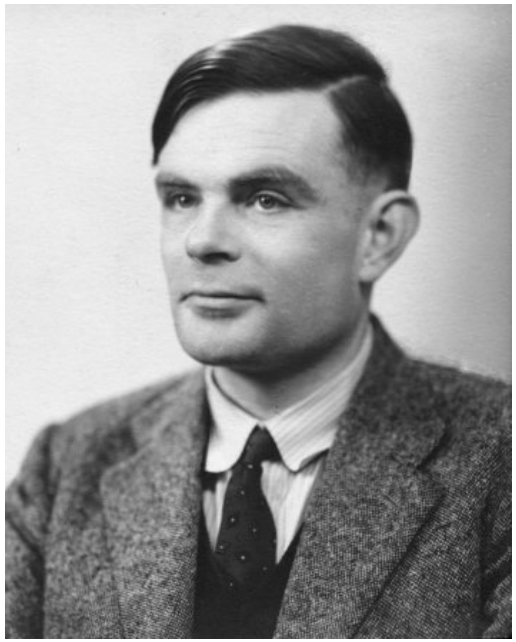
1. Introducción. $\mathcal{P}$ y $\mathcal{NP}$.
2. Máquinas de Turing
3. $\mathcal{NP}$-completitud

¿Por qué máquinas de Turing?

- Para estudiar la complejidad de los problemas se suele usar un modelo abstracto de computador llamado *máquina de Turing*:
  - Es muy sencillo...
  - ...pero tiene la misma capacidad de cómputo que otros sistemas mucho más complejos (todo lo que se puede computar en un lenguaje de programación como Pascal o C se puede computar en una máquina de Turing).

Alan Turing

Alan Turing (1912-1954)



Máquina de Turing (1)

Una máquina de Turing es una máquina que manipula símbolos en una cinta según una tabla de reglas. Se compone de:

- Una *cinta* infinitamente larga (por la derecha) donde se pueden leer/escribir símbolos,
- Una *cabeza lectora/escritora* que apunta a una posición de la cinta y se puede mover,
- Una *control de estado finito*, es decir:
  - Un registro almacena el *estado* en que está la máquina en un momento dado,
  - La máquina utiliza el estado actual, y el símbolo leído de la cinta, para decidir qué hacer, de acuerdo con unas *reglas*.

Máquina de Turing (2)

- Cada una de las reglas es de la siguiente forma: "si el estado actual es q_0 y el símbolo leído por la cabeza lectora es s_0, entonces escribir el símbolo s_1 en la cinta, mover la cabeza lectora hacia [*izquierda*|*derecha*], y pasar al estado q_1".
- Según las reglas que fijemos, podemos conseguir que la máquina de Turing calcule distintas funciones.
- Con este mecanismo tan simple podemos computar cualquier función que podamos computar en un lenguaje de programación (C, Java, etc.) Hay distintas variantes, todas equivalentes (cinta infinita hacia los dos lados, varias cintas, posibilidad de que la cabeza se quede en el sitio, etc.)

Máquina de Turing: ejemplo 1 (1)

Determinar si la longitud de una cadena es par o impar.

- Estados: $\{q_0, q_1, q_P^f, q_I^f\}$
- Alfabeto (símbolos que pueden aparecer en la cinta): $\{0, 1, b\}$
  - A b lo llamaremos el *símbolo en blanco*. Consideraremos que la cinta siempre empieza por un símbolo en blanco (marca de inicio), a continuación tiene la cadena que se quiere procesar, y el resto de la cinta está relleno de símbolos en blanco. Suponemos también que la cabeza lectora empieza sobre el primer símbolo de la cadena (sin contar el símbolo en blanco que marca el inicio).

Máquina de Turing: ejemplo 1 (2)

Determinar si la longitud de una cadena es par o impar. Reglas:

Si el estado es	y leemos un	: escribir un	, pasar al estado	y mover hacia
q_0	0	0	q_1	$\rightarrow$
q_0	1	1	q_1	$\rightarrow$
q_1	0	0	q_0	$\rightarrow$
q_1	1	1	q_0	$\rightarrow$
q_0	b	b	q_P^f	$=$
q_1	b	b	q_I^f	$=$

- Esta máquina indica si la cadena es par o impar según si termina su ejecución en q_P^f (estado de aceptación) o q_I^f (al no haber reglas para estos estados, se termina la ejecución al llegar a ellos).
- Equivalentemente, podríamos borrar la cadena de la cinta y escribir 0 ó 1 para indicar par o impar.

Máquina de Turing: ejemplo 2 (1)

Determinar si una cadena es un palíndromo (se lee igual al derecho que al revés)

- Estados:
  $\{Lee1o, Leido0, Leido1, Comprobar0, Comprobar1, Volver, Aceptar\}$
- Alfabeto (símbolos que pueden aparecer en la cinta): $\{0, 1, b\}$

Máquina de Turing: ejemplo 2 (2)

Determinar si una cadena es un palíndromo. Reglas:

Si el estado es	y leemos un	: escribir un	, pasar al estado	y mover hacia
Lee1o	0	*b*	*RecienLeido0*	$\rightarrow$
Lee1o	1	*b*	*RecienLeido1*	$\rightarrow$
Lee1o	*b*	*b*	*Aceptar*	$=$
RecienLeido0	0/1	0/1	*Leido0*	$\rightarrow$
RecienLeido1	0/1	0/1	*Leido1*	$\rightarrow$
RecienLeido0	*b*	*b*	*Aceptar*	$=$
RecienLeido1	*b*	*b*	*Aceptar*	$=$
Leido0	0/1	0/1	*Leido0*	$\rightarrow$
Leido1	0/1	0/1	*Leido1*	$\rightarrow$
Leido0	*b*	*b*	*Comprobar0*	$\leftarrow$
Leido1	*b*	*b*	*Comprobar1*	$\leftarrow$
Comprobar0	0	*b*	*Volver*	$\leftarrow$
Comprobar1	1	*b*	*Volver*	$\leftarrow$
Volver	0/1	0/1	*Volver*	$\leftarrow$
Volver	*b*	*b*	*Leer1o*	$\rightarrow$

- Si la cadena es un palíndromo, terminamos en el estado *Aceptar*.

Máquina de Turing (3)

- Estas máquinas de Turing reconocen determinados *lenguajes*, es decir, conjuntos de cadenas.
  - Lenguaje formado por las cadenas de longitud par.
  - Lenguaje formado por los palíndromos.
- Estructuras de datos como arrays, árboles, grafos... también se pueden representar como cadenas, así que las máquinas de Turing se pueden usar en problemas con estructuras de datos complejas.
- Todos los problemas con respuesta sí/no se pueden ver como problemas de determinar si una cadena pertenece a un lenguaje (p.ej. el conjunto de las cadenas que representan conjuntos de enteros que tienen un subconjunto que suma 0).

$\mathcal{P}$ y $\mathcal{NP}$ con máquinas de Turing

La definición original de $\mathcal{P}$ y $\mathcal{NP}$ es:

- Clase de complejidad $\mathcal{P}$: Conjunto de lenguajes (o sea, problemas sí/no) que se pueden resolver con una máquina de Turing que termina su ejecución en un número de pasos acotado por una función polinómica, $O(n^k)$.
- Clase de complejidad $\mathcal{NP}$: Conjunto de lenguajes (o sea, problemas sí/no) que se pueden resolver con una máquina de Turing no determinista que termina su ejecución en un número de pasos acotado por una función polinómica, $O(n^k)$.

Una máquina de Turing no determinista es una que puede tomar varios caminos *a la vez* ante una situación dada:

Si el estado es	y leemos un	: escribir un	, pasar al estado	y mover hacia
q_1	0	q_2	0	$\rightarrow$
q_1	0	q_3	1	$\leftarrow$

$\mathcal{P}$ y $\mathcal{NP}$ con máquinas de Turing (2)

- Todo lo que se puede computar con una máquina de Turing no determinista se puede hacer también en una determinista...
  - ...pero no con la misma eficiencia, por supuesto: de ahí que pueda haber problemas que están en $\mathcal{NP}$ pero no en $\mathcal{P}$.
- Los ordenadores que nosotros manejamos son deterministas.
  - Lo análogo a una máquina de Turing no determinista "en la vida real" sería un procesador con una cantidad infinita de núcleos. Por supuesto, no existe (si existiera, sería fácil escribir un algoritmo multihilo que resolviese el problema de la suma de subconjuntos en tiempo polinómico).

Outline

1. Introducción. $\mathcal{P}$ y $\mathcal{NP}$.
2. Máquinas de Turing
3. $\mathcal{NP}$-completitud

Problemas $\mathcal{NP}$-completos

Existe un subconjunto de problemas en $\mathcal{NP}$, llamados problemas $\mathcal{NP}-$*completos*, que tienen la siguiente curiosa propiedad:

- Si cualquiera de ellos está en $\mathcal{P}$, entonces todos los problemas de $\mathcal{NP}$ están en $\mathcal{P}$. Es decir, bastaría con encontrar un algoritmo polinómico para *uno solo* de ellos para demostrar que $\mathcal{P} = \mathcal{NP}$.
- De lo anterior se deduce que, si $\mathcal{P} \neq \mathcal{NP}$, *ninguno* de estos problemas puede estar en $\mathcal{P}$. Es decir, si se demostrara que $\mathcal{P} \neq \mathcal{NP}$, sabríamos de un golpe que para todos estos problemas (que son muchos) nunca encontraremos una solución polinómica. (y, dado que se han encontrado varios miles de problemas $\mathcal{NP}$-completos y muchos de ellos tienen gran relevancia práctica, esto hace ver por qué el problema de $\mathcal{P}$ y $\mathcal{NP}$ se considera uno de los "problemas del milenio"...)

¿Cómo se ha llegado a estas conclusiones?

Reducibilidad

- Decimos que un problema A es *reducible en tiempo polinómico* a un problema B si, dado un algoritmo que resuelve B, podemos obtener un algoritmo que resuelve A añadiendo, como mucho, un factor polinómico a su complejidad.
  - Intuitivamente: construimos un algoritmo para A haciendo una llamada a función al algoritmo para B.
  - Esto quiere decir que, en términos de complejidad computacional, A es al menos tan "fácil" como B (por ejemplo, si B estaba en $\mathcal{P}$, A también estará en $\mathcal{P}$).
- Un problema es $\mathcal{NP}$-completo si está en $\mathcal{NP}$ y cualquier problema en $\mathcal{NP}$ es reducible a él.
  - Por eso si un problema $\mathcal{NP}$-completo estuviera en $\mathcal{P}$, todos los problemas de $\mathcal{NP}$ estarían en $\mathcal{P}$ (serían reducibles a él).

Ejemplos de problemas $\mathcal{NP}$-completos

Algunos ejemplos de problemas $\mathcal{NP}$-completos:

- Suma de subconjuntos.
- Problema del viajante: recorrer todas las ciudades de un mapa sin repetir ninguna, y volviendo al punto de partida.
- Determinar si los países de un mapa se pueden colorear con k colores ($k > 4$) sin que países adyacentes tengan el mismo color.
- Dada una lista de personas y sus enemistades, ¿cuál es el máximo número de personas que podemos invitar a una boda sin que nadie se lleve mal con nadie?
- Satisfacibilidad booleana: determinar si una expresión de lógica proposicional (como las condiciones de un if) es verdadera para alguna combinación de valores de las variables
- Resolver un sudoku.
- Etc...

Más sobre el problema de $\mathcal{P}$ y $\mathcal{NP}$

Según lo que hemos visto, bastaría con encontrar una solución polinómica a uno cualquiera de estos problemas $\mathcal{NP}$-completos para demostrar que $\mathcal{P} = \mathcal{NP}$.

- Que tras tantos años intentándolo con tantos problemas distintos, no se haya logrado, es una de las razones por las que muchos investigadores piensan que $\mathcal{P} \neq \mathcal{NP}$.
- Otro argumento es más filosófico: Si $\mathcal{P} = \mathcal{NP}$, el mundo sería muy diferente de como normalmente suponemos que es. Los "saltos creativos" no tendrían ningún valor especial, no habría diferencias importantes entre resolver un problema y reconocer la solución una vez encontrada. Cualquiera que pudiese apreciar una sinfonía sería Mozart, cualquiera que pudiese entender una demostración paso a paso sería Gauss, cualquiera que pudiese reconocer una buena estrategia de inversión sería Warren Buffett..." (Scott Aaronson, MIT)