Problemas Intratables

Horst H. von Brand
vonbrand@inf.utfsm.cl

Departamento de Informática
Universidad Técnica Federico Santa María

Contenido

Problemas difíciles

Hay muchos problemas para los que se conocen algoritmos eficientes, incluso elegantes. Hay otros, para los cuales luego de siglos de esfuerzo solo hay soluciones en casos muy especiales, soluciones engorrosas del tipo «pruebe todas las posibilidades» o métodos no mucho mejores que lo anterior.

Hay muchos problemas para los cuales hay algoritmos eficientes para verificar una solución propuesta, pero no se conocen métodos razonables que construyan tal solución.
Daremos un vistazo a lo que se sabe de este fenómeno.

Algunos problemas

Camino más corto: Dados los tiempos necesarios para viajar entre las distintas ciudades de Chile, encuentre el camino más rápido para llegar de Arica a Punta Arenas.

Camino más largo: Encuentre el camino más lento sin repetir pasos intermedios para llegar de Arica a Punta Arenas.

Vendedor viajero: Encuentre el camino más rápido que permita visitarlas todas volviendo al punto de partida.

En los tres casos hay una enorme cantidad de caminos posibles. El primero tiene solución eficiente (como el algoritmo de Dijkstra), para el segundo y el tercero no se conoce un algoritmo esencialmente mejor que probar todas las opciones.

Problemas de búsqueda y decisión

Otro problema que no tiene solución eficiente conocida es el problema SAT: dada la expresión lógica $\phi(x_1,\ldots,x_n)$ en n variables, ¿qué valores de las variables x_k que la hacen verdadera?
Distinguiremos dos variantes de este problema:

Búsqueda: Dada $\phi(x_1,\ldots,x_n)$, dé valores de $x_1,\ldots,x_n$ que la hagan verdadera, o indique que no es posible.

Decisión: Dada $\phi(x_1,\ldots,x_n)$, indique si hay valores de $x_1,\ldots,x_n$ que la hagan verdadera.

Problemas de búsqueda y decisión

Búsqueda y decisión parecen diferentes. Si tenemos cómo resolver el primer problema, es simple resolver el segundo.

Supongamos que nos proveen una subrutina para el problema de decisión, $M(\phi(x_1,\ldots,x_n))$. Si $M(\phi(x_1,\ldots,x_n))$ nos dice que no puede satisfacerse $\phi(x_1,\ldots,x_n)$, esa es nuestra respuesta final. Si hay solución, asigne $x_1 \leftarrow M(\phi(\text{True},x_2,\ldots,x_n))$ y continúe con x_2 como primera variable en $\phi(x_1,x_2,\ldots,x_n)$ (ya x_1 tiene un valor). En cada ronda asignamos valor a una variable, en n rondas hallamos una asignación de valores que hacen verdadera $\phi(x_1,\ldots,x_n)$.

Problemas de búsqueda y decisión

Resulta que en muchos de los problemas de interés el problema de búsqueda se reduce al de decisión, en el sentido que podemos resolver el problema de búsqueda usando juiciosamente resultados para problemas de decisión, como en el ejemplo de SAT dado antes.

Problemas de decisión son más simples de tratar, podemos asociarlos al lenguaje de las palabras para las que la respuesta es «Sí», y usar la maquinaria de autómatas desarrollada antes. Es más fácil razonar con simples respuestas «Sí» y «No».

Problemas

Llamaremos *problema* a un lenguaje. La respuesta a una instancia del problema (una palabra sobre el alfabeto adecuado) es solo «Sí» (la palabra pertenece al lenguaje) o «No» (la palabra no pertenece al lenguaje).

Problemas

Nos interesan problemas (lenguajes) infinitos: si el número de instancias con respuesta «Sí» es finito, basta construir una lista de esas instancias de antemano y revisar si la instancia propuesta aparece en la lista. La misma observación vale si el lenguaje es co-finito (su complemento es finito).

Problemas

Interesa por tanto el comportamiento del algoritmo conforme crece la instancia del problema. Para precisar lo que entendemos por tamaño de la instancia del problema, fijamos un alfabeto, y llamaremos *tamaño* de esa instancia al largo de la palabra que lo describe. Para evitar que se haga trampa, exigiremos alguna representación eficiente (números en binario, no agregar información redundante, ...). Básicamente, como representaríamos los datos en un programa.

¿Qué entendemos por «solución eficiente»?

Discutimos en detalle el modelo de cómputo máquina de Turing. La tesis de Church asevera que es equivalente a nuestra idea de «computación». Otros modelos de cómputo se demostraron equivalentes.

¿Qué entendemos por «solución eficiente»?

Para todos los modelos hay alguna medida natural de «tiempo» (el recurso más importante en la solución de un problema complicado). Por ejemplo, pasos dados por la máquina de Turing, número de instrucciones ejecutadas por la CPU, veces que se llama alguna función en el cálculo lambda, ... Resulta que las medidas de «tiempo» están relacionadas por polinomios: Un programa RASP que ejecuta n instrucciones puede simularse mediante una máquina de Turing que da a lo más cn^4 pasos para una constante c; un paso de una máquina de Turing puede simularse en un número finito de instrucciones RASP. Relaciones similares existen entre los otros modelos.

¿Qué entendemos por «solución eficiente»?

Si el tiempo de ejecución de un algoritmo crece más rápido que todos los polinomios, difícilmente lo llamaremos «eficiente».
Esto sugiere:

Definición

Un algoritmo se llamará *eficiente* si su tiempo de ejecución está acotado por un polinomio en el tamaño de los datos de entrada.

Los polinomios son cerrados respecto de composición, lo que con la afortunada coincidencia de la relación polinomial entre modelos de cómputo nos permite suponer cualquier modelo de cómputo a la hora de discutir eficiencia.

Problemas en P y en NP

Vimos que en principio las máquinas de Turing deterministas y no-deterministas son equivalentes: reconocen el mismo conjunto de lenguajes. Pero nuestra simulación determinista de una máquina de Turing no-determinista tiene costo exponencial: revisa *todas* las posibles computaciones, siguiendo cada alternativa.

La diferencia parece ser fundamental, todo indica que no hay manera de evitar ese aumento.

Problemas en P y en NP

Definición

Un problema se dice que está en P (tiempo determinista polinomial) si hay una máquina de Turing determinista que lo decide en un número de pasos acotado por un polinomio en el tamaño de la instancia.

Definición

Un problema se dice que está en NP (tiempo no-determinista polinomial) si hay una máquina de Turing que lo decide en un número de pasos acotado por un polinomio en el tamaño de la instancia.

En el caso de un autómata no-determinista, el largo de la computación es del camino más corto que lleva a aceptar.

Observaciones

Es claro que $P \subseteq NP$. La pregunta de si $P = NP$ es uno de los problemas abiertos famosos en matemáticas, y definitivamente central en computación. La evidencia es que no son iguales: hay bastantes problemas de inmenso interés práctico en NP, algunos de los cuales se han estudiado por décadas sin hallar algoritmos deterministas eficientes. (Es preferible pensar que no hay algoritmos polinomiales a reconocer que como especie somos demasiado tontos para hallarlos. . .)

Son un subconjunto de los problemas decidibles (simule la máquina de Turing no-determinista del caso por el máximo número de pasos en que promete hallar la respuesta, si no ha respondido «Sí» en ninguna de las ramas de cómputo, la respuesta debe ser «No»).

Nuestra situación

Sospechamos $P \neq NP$, queremos razonar acerca de problemas en NP que no están en P, pero ni siquiera sabemos si tal cosa existe. Definiremos un conjunto de problemas que en cierto sentido son los más difíciles de NP, concentrándonos en estudiar esos. (Es de esperar que no llegue un aguafiestas y demuestre $P = NP$, echando a perder todo.)

Definiciones

Definición

Se dice que A *se reduce polinomialmente a* B (se anota $A \leq_p B$) si hay un algoritmo determinista que traduce toda instancia σ de A en una instancia de B con la misma respuesta en tiempo polinomial.

Definiciones

Igual que antes, solo que ahora exigimos que la traducción se efectúe en tiempo polinomial en el tamaño de la instancia original. Note que el tamaño de la instancia resultante de B está acotada por un polinomio en el tamaño de la instancia de A. Si B tiene solución en tiempo polinomial, podemos usarla con la reducción para construir un algoritmo polinomial para A. Nuevamente, $\leq_p$ es un preorden (relación transitiva y reflexiva).
La relación $\leq_p$ es casi un orden parcial: es reflexiva y transitiva, pero no antisimétrica. Demostrar esto queda de ejercicio.

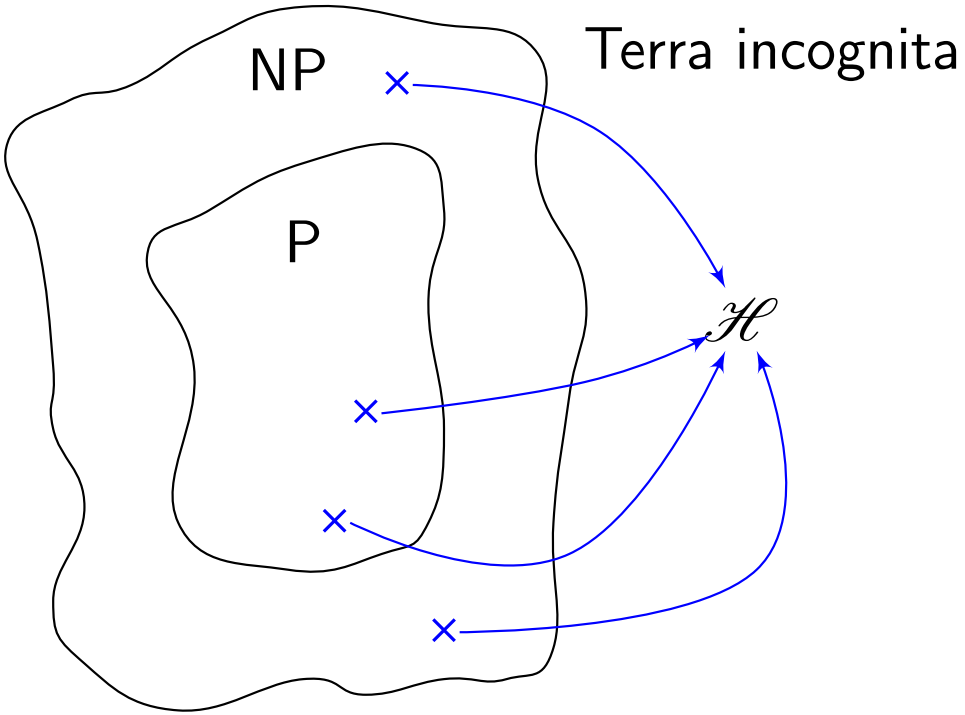
Definiciones

Definición

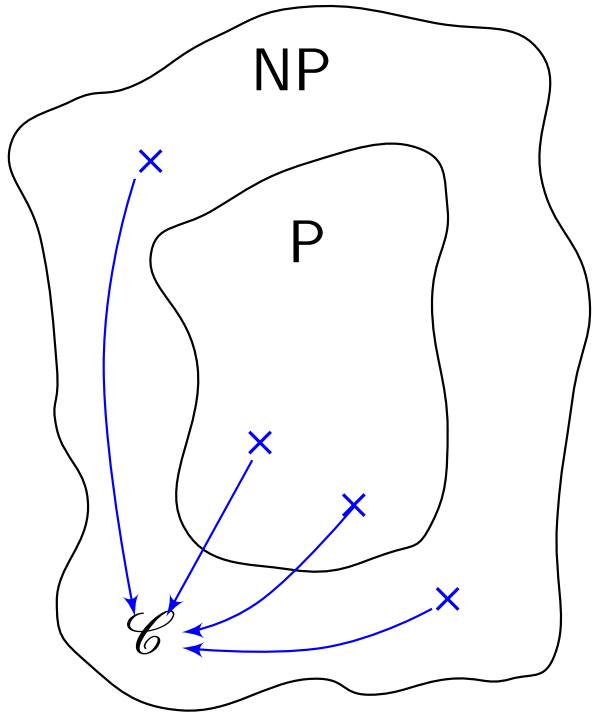
Un problema se dice NP-*duro* si todos los problemas en NP se reducen polinomialmente a él. Un problema se dice NP-*completo* si es NP-duro y está en NP.

A la hora de demostrar que un problema es NP-completo, normalmente es obvio que el problema está en NP, la complicación es demostrar que es NP-duro.

Problemas en P, en NP y NP-completos



(a) $\mathscr{H}$ es NP-duro



(b) $\mathscr{C}$ es NP-completo

Demostrar NP-duro

Teorema

Si $\mathscr{N}$ es un problema NP*-completo, y $\mathscr{N} \leq_p \mathscr{H}$, entonces $\mathscr{H}$ es* NP*-duro.*

Demostración.

Por definición, para todo problema $\mathscr{A} \in$ NP es $\mathscr{A} \leq_p \mathscr{N}$ (como $\mathscr{N}$ es NP-completo, es NP-duro), con $\mathscr{N} \leq_p \mathscr{H}$ tenemos $\mathscr{A} \leq_p \mathscr{H}$. □

Definición alternativa de NP

Teorema

El problema $\mathscr{P}$ está en NP *si y solo si para cada instancia $\sigma \in \mathscr{P}$ hay un certificado c tal que una máquina de Turing determinista con entrada σ y c verifica que $\sigma \in \mathscr{P}$ en tiempo polinomial en $|\sigma|$.*

Definición alternativa de NP

Demostración.

Si $\mathscr{P}$ está en NP, hay un polinomio p y una máquina de Turing no-determinista M que acepta $\sigma \in \mathscr{P}$ en a lo más $p(|\sigma|)$ pasos. Un certificado para σ es la secuencia de estados de M al aceptar σ.

Si $\mathscr{P}$ cumple con lo indicado por el teorema, podemos construir una máquina de Turing que acepta $\sigma \in \mathscr{P}$ en tiempo polinomial a través de generar no-deterministamente un certificado c para σ, para luego verificar usando la máquina de Turing dada que $\sigma \in \mathscr{P}$.

Como la verificación toma tiempo acotado por un polinomio en $|\sigma|$, el largo leído de c no puede ser más que esto, y puede generarse en tiempo polinomial. □

Definiciones

Muy bonito todo esto, pero hace falta un problema NP-completo para echar a andar la maquinaria. Hay uno obvio, casi por definición: el problema NTM da una máquina de Turing M, un polinomio p y una palabra σ, y pregunta si M acepta σ en a lo más $p(|\sigma|)$ pasos.

Pero quedamos en las mismas, expresar la computación de una máquina de Turing en términos de un problema cualquiera (para reducir) es una soberana lata.

Fórmulas lógicas

Una fórmula lógica es una expresión formada por variables y las conocidas operaciones. Una pregunta obvia es si hay una asignación de valores de verdad a las variables que hace verdadera la fórmula. Este es el problema SAT (por *satisfability*).

Fórmulas lógicas en forma normal conjuntiva

Usando las equivalencias:

$$P \iff Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$
$$P \Rightarrow Q \equiv \neg P \vee Q$$
$$(P \vee Q) \wedge R \equiv (P \wedge R) \vee (Q \wedge R)$$
$$(P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)$$

junto con de Morgan, podemos eliminar paréntesis y negaciones, terminando con una expresión que es la conjunción de cláusulas, cada una de las cuales es la disyunción de literales, donde un literal es una variable o su negación.

Para abreviar, anotaremos $\overline{x}$ para la negación de x en lo que sigue.

Fórmulas lógicas en forma normal conjuntiva

A una fórmula así se le llama en *forma normal conjuntiva* (abreviada CNF, por *conjunctive normal form*). Un ejemplo de fórmula en CNF es:

$$\phi(x_1, x_2, x_3, x_4, x_5) = (x_1 \vee \overline{x}_5) \wedge x_2 \wedge (x_1 \vee \overline{x}_2 \vee x_4 \vee x_5) \wedge (\overline{x}_3 \vee \overline{x}_4)$$

La transformación puede hacerse en tiempo polinomial en el largo de la fórmula original.

El problema CSAT

El problema CSAT da una fórmula en forma normal conjuntiva y pregunta si puede satisfacerse. Por lo anterior sabemos que

SAT $\leq_p$ CSAT, además que por ser CSAT un caso particular de SAT es claro que CSAT $\leq_p$ SAT.

El problema k-SAT

Una fórmula lógica se dice que está en k-CNF si está en forma normal conjuntiva y cada cláusula tiene un máximo de k literales. El problema k-SAT da una fórmula lógica en k-CNF y pregunta si es satisfacible.

El problema 1-SAT tiene solución obvia: solo si aparece una variable y su negación la fórmula no es satisfacible.

El problema 2-SAT tiene solución en tiempo lineal en el largo de la fórmula, pero no nos detendremos en eso.

Toda fórmula puede llevarse a 3-CNF

Dada una fórmula en CNF, si hay una cláusula con más de 3 literales, digamos:

$$x_1 \vee x_2 \vee \cdots \vee x_n$$

podemos reescribirla usando variables auxiliares $y_1, y_2, \ldots, y_{n-3}$ como:

$$(x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge (\overline{y}_2 \vee x_4 \vee y_3) \wedge \cdots \wedge (\overline{y}_{n-3} \vee x_{n-1} \vee x_n)$$

La segunda es satisfacible si y solo si lo es la primera.
Esta traducción podemos hacerla en tiempo lineal mediante un programa en un lenguaje como Python.

3-CNF

Podemos exigir que toda cláusula tenga exactamente tres literales rellenando con variables auxiliares. Esta restricción puede simplificar demostraciones.
Por ejemplo:

$$x$$

da:

$$(x \vee y) \wedge (x \vee \overline{y})$$

y

$$x \vee \overline{y}$$

da:

$$(x \vee \overline{y} \vee z) \wedge (x \vee \overline{y} \vee \overline{z})$$

3-CNF

Esta traducción nuevamente podemos hacerla en tiempo lineal mediante un programa en un lenguaje como Python.

El problema 3SAT

El problema 3SAT pone una fórmula lógica en 3CNF y pregunta si es satisfacible.

Reducciones entre formas

Tenemos la siguiente cadena de reducciones (que no demostraremos en detalle):

$$\text{SAT} \leq_p \text{CSAT} \leq_p \text{3SAT}$$

3SAT es importante porque pone restricciones que simplifican reducir desde este problema.

El teorema de Cook-Levin

Teorema (Cook-Levin)

SAT *es* NP-*completo.*

El teorema de Cook-Levin

La demostración es un tanto larga, no la detallaremos acá.
Un ingrediente clave es que tenemos variables $x_1, x_2, \ldots, x_n$ y queremos expresar que exactamente una de ellas toma el valor verdadero. Ilustramos el forzar que exactamente una variable sea verdadera mediante un ejemplo. Supongamos variables x, y, z, usamos:

$$(x \vee y \vee z) \wedge (\overline{x} \vee \overline{y}) \wedge (\overline{x} \vee \overline{z}) \wedge (\overline{y} \vee \overline{z})$$

Agregamos n cláusulas que solo dejan una variable sin negar, cada una de $n-1$ literales. En total son $n + n(n-1) = n^2$ literales.

El teorema de Cook-Levin

La idea es tener una colección de variables, contenido por posición en la cinta (nunca se consideran más que $p(|\sigma|)$ de ellas) y paso de la máquina (nunca da más de $p(|\sigma|)$ pasos) que expresan que en el paso k en la posición i está un símbolo dado, que el cabezal está en este casillero, que en el paso k el estado es q. Usamos la idea anterior para expresar que el contenido de un casillero en un momento dado es uno solo, y así sucesivamente. Fórmulas adicionales relacionan el contenido de la cinta en el paso k con el paso $k+1$. Además exigimos que el estado al terminar el cómputo sea final.

El teorema de Cook-Levin

Lo anterior demuestra que SAT es NP-duro. Para demostrar que está en NP un certificado es valores de las variables que hacen verdadera la fórmula.

3SAT

Corolario

3SAT *es* NP-*completo.*

Demostración.

3SAT es un caso particular de SAT, con lo que 3SAT está en NP, y vimos que SAT $\leq_p$ 3SAT. □

Resumen

- Discutimos problemas de búsqueda y decisión.
- Definimos *algoritmos eficientes*.
- Definimos *reducción polinomial*.
- Definimos las clases de problemas P y NP. Es obvio que $\mathsf{P} \subseteq \mathsf{NP}$. ¿Es $\mathsf{P} = \mathsf{NP}$?
- Definimos problemas NP-duros y NP-completos.
- Para demostrar que un problema A está en NP, para $\sigma \in A$ demuestre que hay un *certificado* c tal que hay un algoritmo eficiente que demuestre que $\sigma \in A$ dados σ, c.
- Para demostrar que el problema A es NP-duro, basta hallar un problema NP-completo C tal que $C \leq_p A$.