Complejidad Computacional

José de Jesús Lavalle Martínez

Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación
Maestría en Ciencias de la Computación
Análisis y Diseño de Algoritmos
MCOM 20300

Contenido

Definición 1

Un lenguaje $L \subseteq \Sigma^*$ es **decidible** si y sólo si existe una máquina de Turing T que lo reconoce, es decir, para toda $w \in \Sigma^*$ la máquina de Turing T regresa **sí** cuando $w \in L$ y regresa **no** si $w \notin L$.

Preliminares

Si bien el concepto de Máquina de Turing es la formalización de lo que es computable, de igual manera puede considerar un programa en cualquier lenguaje de programación que reconozca el lenguaje L.

Definición 2

Sea $t : \mathbb{N} \to \mathbb{R}^+$ una función. Defina la **clase de complejidad time**, en símbolos $\mathrm{TIME}(t(n))$, como la colección de todos los lenguajes que son decidibles por una máquina de Turing en tiempo $O(t(n))$.

La clase P I

- Para nuestros propósitos, las diferencias polinomiales en el tiempo de ejecución se consideran pequeñas, mientras que las diferencias exponenciales se consideran grandes.

La clase P I

- Veamos por qué elegimos hacer esta separación entre polinomios y exponenciales en lugar de entre algunas otras clases de funciones.

La clase P I

- Primero, observe la gran diferencia entre la tasa de crecimiento de polinomios típicos como n^3 y exponenciales que ocurren típicamente como 2^n.

La clase P I

- Por ejemplo, sea $n = 1000$, el tamaño de una entrada razonable para un algoritmo.

La clase P I

- En ese caso, n^3 es mil millones, un número grande pero manejable, mientras que 2^n es un número mucho mayor que el número de átomos en el universo.

La clase P I

- Los algoritmos de tiempo polinomial son lo suficientemente rápidos para muchos propósitos, pero los algoritmos de tiempo exponencial rara vez son útiles.

La clase P II

- Los algoritmos de tiempo exponencial surgen típicamente cuando resolvemos problemas buscando exhaustivamente en un espacio de soluciones, llamado **búsqueda de fuerza bruta**.

La clase P II

- Por ejemplo, una forma de factorizar un número en sus primos constituyentes es buscar en todos los divisores potenciales.

La clase P II

- El tamaño del espacio de búsqueda es exponencial, por lo que esta búsqueda utiliza tiempo exponencial.

La clase P II

- A veces, la búsqueda por fuerza bruta puede evitarse mediante una comprensión más profunda de un problema, lo que puede revelar un algoritmo de tiempo polinomial de mayor utilidad.

La clase P II

- Todos los modelos computacionales deterministas razonables son **polinomialmente equivalentes**.

La clase P II

- Es decir, cualquiera de ellos puede simular otro con sólo un aumento polinomial en el tiempo de ejecución.

La clase P III

- Cuando decimos que todos los modelos deterministas razonables son polinomialmente equivalentes, no intentamos definir lo razonable.

La clase P III

- Sin embargo, tenemos en mente una noción lo suficientemente amplia como para incluir modelos que se aproximan mucho a los tiempos de ejecución en computadoras reales.

La clase P III

- A partir de aquí, nos centraremos en aspectos de la teoría de la complejidad del tiempo que no se ven afectados por las diferencias polinomiales en el tiempo de ejecución.

La clase P III

- Ignorar estas diferencias nos permite desarrollar una teoría que no depende de la selección de un modelo particular de computación.

La clase P III

- Recuerde, nuestro objetivo es presentar las propiedades fundamentales de la *computación*, más que las propiedades de las máquinas de Turing o cualquier otro modelo especial de cómputo, como puede ser algún otro lenguaje de programación.

La clase P IV

- Puede sentir que ignorar las diferencias polinomiales en el tiempo de ejecución es absurdo.

La clase P IV

- Los programadores reales ciertamente se preocupan por estas diferencias y trabajan duro sólo para que sus programas se ejecuten el doble de rápido.

La clase P IV

- Sin embargo, ignoramos los factores constantes hace un tiempo cuando introdujimos la notación asintótica.

La clase P IV

- Ahora proponemos ignorar las diferencias polinomiales mucho mayores, como la que existe entre el tiempo n y n^3.

La clase P IV

- Nuestra decisión de ignorar las diferencias polinomiales no implica que consideremos tales diferencias sin importancia.

La clase P IV

- Por el contrario, ciertamente consideramos que la diferencia entre el tiempo n y el tiempo n^3 es importante.

La clase P V

- Pero algunas preguntas, como la polinomialidad o no polinomialidad del problema de factorización, también son importantes y no dependen de diferencias polinomiales.

La clase P V

- Simplemente elegimos enfocarnos en este tipo de pregunta aquí. Ignorar los árboles para ver el bosque no significa que uno sea más importante que el otro, solo da una perspectiva diferente.

La clase P V

- Ahora llegamos a una definición importante en la teoría de la complejidad.

Definición 3

P es la clase de lenguajes que son decidibles en tiempo polinomial por una Máquina de Turing determinista con una sola cinta. En otras palabras,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

Definición 3

P es la clase de lenguajes que son decidibles en tiempo polinomial por una Máquina de Turing determinista con una sola cinta. En otras palabras,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

La clase P juega un papel central en nuestra teoría y es importante porque:

Definición 3

P es la clase de lenguajes que son decidibles en tiempo polinomial por una Máquina de Turing determinista con una sola cinta. En otras palabras,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

La clase P juega un papel central en nuestra teoría y es importante porque:

1. P es invariante para todos los modelos de cálculo que son polinomialmente equivalentes a la máquina de Turing determinista de una sola cinta, y

Definición 3

P es la clase de lenguajes que son decidibles en tiempo polinomial por una Máquina de Turing determinista con una sola cinta. En otras palabras,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

La clase P juega un papel central en nuestra teoría y es importante porque:

1. P es invariante para todos los modelos de cálculo que son polinomialmente equivalentes a la máquina de Turing determinista de una sola cinta, y
2. P corresponde aproximadamente a la clase de problemas que se pueden resolver de manera realista en una computadora.

La clase P VII

- El ítem 1 indica que P es una clase matemáticamente robusta.

La clase P VII

- No se ve afectado por los detalles del modelo de cálculo que estamos usando.

La clase P VII

- El ítem 2 indica que P es relevante desde un punto de vista práctico.

La clase P VII

- Cuando un problema está en P, tenemos un método para resolverlo que se ejecuta en el tiempo n^k para alguna constante k.

La clase P VII

- Si este tiempo de ejecución es práctico depende de k y de la aplicación.

La clase P VIII

- Por supuesto, es poco probable que un tiempo de ejecución de n^{100} sea de utilidad práctica.

La clase P VIII

- No obstante, se ha demostrado que es útil llamar al tiempo polinomial el umbral de la capacidad de solución práctica.

La clase P VIII

- Una vez que se ha encontrado un algoritmo de tiempo polinomial para un problema que anteriormente parecía requerir un tiempo exponencial, se ha obtenido una visión clave de él y, por lo general, se siguen reducciones adicionales en su complejidad, a menudo hasta el punto de una utilidad práctica real.

Ejemplos de problemas en P I

- Cuando presentamos un algoritmo de tiempo polinomial, damos una descripción de alto nivel sin hacer referencia a las características de un modelo computacional en particular.

Ejemplos de problemas en P I

- Al hacerlo, se evitan los tediosos detalles de las cintas y los movimientos de la cabeza de una máquina de Turing.

Ejemplos de problemas en P I

- Seguimos ciertas convenciones cuando describimos un algoritmo para poder analizar su polinomialidad.

Ejemplos de problemas en P I

- Seguimos describiendo algoritmos con pasos numerados.

Ejemplos de problemas en P I

- Ahora debemos ser sensibles al número de pasos de la máquina de Turing requeridos para implementar cada paso, así como al número total de pasos que usa el algoritmo.

Ejemplos de problemas en P I

- Cuando analizamos un algoritmo para mostrar que se ejecuta en tiempo polinomial, debemos hacer dos cosas.

Ejemplos de problemas en P I

- Primero, tenemos que dar un límite superior polinomial (generalmente en notación O) en el número de pasos que usa el algoritmo cuando se ejecuta en una entrada de longitud n.

Ejemplos de problemas en P II

- Luego, tenemos que examinar los pasos individuales en la descripción del algoritmo para estar seguros de que cada uno puede implementarse en tiempo polinomial en un modelo determinista razonable.

Ejemplos de problemas en P II

- Elegimos los pasos cuando describimos el algoritmo para que esta segunda parte del análisis sea fácil de hacer.

Ejemplos de problemas en P II

- Cuando se han completado ambas tareas, podemos concluir que el algoritmo se ejecuta en tiempo polinomial porque hemos demostrado que se ejecuta para un número polinomial de pasos, cada uno de los cuales se puede realizar en tiempo polinomial, y la composición de polinomios es una polinomio.

Ejemplos de problemas en P II

- Un punto que requiere atención es el método de codificación utilizado para los problemas.

Ejemplos de problemas en P II

- Continuamos usando la notación de corchetes angulares $\langle \cdot \rangle$ para indicar una codificación razonable de uno o más objetos en una cadena, sin especificar ningún método de codificación en particular.

Ejemplos de problemas en P III

- Ahora bien, un método razonable es el que permite la codificación y decodificación de objetos en tiempo polinomial en representaciones internas naturales o en otras codificaciones razonables.

Ejemplos de problemas en P III

- Los métodos de codificación familiares para grafos, autómatas y similares son razonables.

Ejemplos de problemas en P III

- Pero tenga en cuenta que la notación unaria para codificar números (como en el número 17 codificado por la cadena unaria 11111111111111111) no es razonable porque es exponencialmente más grande que las codificaciones verdaderamente razonables, como la notación base k para cualquier $k \geq 2$.

Ejemplos de problemas en P IV

- Muchos problemas computacionales que encontrará en este capítulo contienen codificaciones de grafos.

Ejemplos de problemas en P IV

- Una codificación razonable de un grafo es una lista de sus nodos y aristas.

Ejemplos de problemas en P IV

- Otra es la matriz de adyacencia, donde la entrada (i, j)-ésima es 1 si hay un arista del nodo i al nodo j y 0 si no la hay.

Ejemplos de problemas en P IV

- Cuando analizamos algoritmos de grafos, el tiempo de ejecución se puede calcular en términos del número de nodos en lugar del tamaño de la representación del grafo.

Ejemplos de problemas en P IV

- En representaciones razonables de grafos, el tamaño de la representación es un polinomio en el número de nodos.

Ejemplos de problemas en P IV

- Así, si analizamos un algoritmo y mostramos que su tiempo de ejecución es polinomial (o exponencial) en el número de nodos, sabemos que es polinomial (o exponencial) en el tamaño de la entrada.

El primer problema se refiere a los grafos dirigidos. Un grafo dirigido G contiene los nodos s y t. El problema $PATH$ consiste en determinar si existe una ruta dirigida de s a t.

Sea

$PATH = \{\langle G, s, t\rangle | G$ es un grafo dirigido que tiene un camino dirigido de s a $t\}$

Sea

$PATH = \{\langle G, s, t\rangle | G$ es un grafo dirigido que tiene un camino dirigido de s a $t\}$

Teorema 4

$PATH \in \mathrm{P}$.

IDEA DE LA PRUEBA

- Demostramos este teorema presentando un algoritmo de tiempo polinomial que decide $PATH$.

IDEA DE LA PRUEBA

- Antes de describir ese algoritmo, observemos que un algoritmo de fuerza bruta para este problema no es lo suficientemente rápido.

IDEA DE LA PRUEBA

- Un algoritmo de fuerza bruta para $PATH$ procede al examinar todas las rutas potenciales en G y determinar si alguna es una ruta dirigida de s a t.

IDEA DE LA PRUEBA

- Una ruta potencial es una secuencia de nodos en G que tiene una longitud de como máximo m, donde m es el número de nodos en G (si existe alguna ruta dirigida de s a t, existe una que tenga una longitud de como máximo m porque nunca es necesario repitir un nodo).

IDEA DE LA PRUEBA

- Pero el número de tales caminos potenciales es aproximadamente m^m, que es exponencial en el número de nodos en G. Por lo tanto, este algoritmo de fuerza bruta usa tiempo exponencial.

IDEA DE LA PRUEBA

- Para obtener un algoritmo de tiempo polinomial para $PATH$, debemos hacer algo que evite la fuerza bruta. Una forma es utilizar un método de búsqueda de grafos como la búsqueda primero en amplitud.

IDEA DE LA PRUEBA

- Aquí, marcamos sucesivamente todos los nodos en G que son accesibles desde s por caminos dirigidos de longitud 1, luego 2, luego 3, hasta m. Es fácil limitar el tiempo de ejecución de esta estrategia por un polinomio.

PRUEBA Un algoritmo polinomial M para $PATH$ opera como sigue.

$M =$ "Como entrada recibe $\langle G, s, t \rangle$, donde G es un grafo dirigido con nodos s y t como origen y destino.

$M =$ "Como entrada recibe $\langle G, s, t \rangle$, donde G es un grafo dirigido con nodos s y t como origen y destino.

1. Coloque una marca en el nodo s.

$PATH \in$ P IV

$M =$ "Como entrada recibe $\langle G, s, t \rangle$, donde G es un grafo dirigido con nodos s y t como origen y destino.

1. Coloque una marca en el nodo s.
2. Repita lo siguiente hasta que no se marquen nodos adicionales:

$PATH \in$ P IV

$M =$ "Como entrada recibe $\langle G, s, t \rangle$, donde G es un grafo dirigido con nodos s y t como origen y destino.

1. Coloque una marca en el nodo s.
2. Repita lo siguiente hasta que no se marquen nodos adicionales:
3. Escanee todos las aristas de G. Si se encuentra una arista (a, b) de un nodo marcado a a un nodo no marcado b, marque el nodo b.

$PATH \in$ P IV

$M =$ "Como entrada recibe $\langle G, s, t \rangle$, donde G es un grafo dirigido con nodos s y t como origen y destino.

1. Coloque una marca en el nodo s.
2. Repita lo siguiente hasta que no se marquen nodos adicionales:
3. Escanee todos las aristas de G. Si se encuentra una arista (a, b) de un nodo marcado a a un nodo no marcado b, marque el nodo b.
4. Si t está marcado *acepta*. De lo contrario *rechaza*".

- Ahora analizamos este algoritmo para demostrar que se ejecuta en tiempo polinomial. Obviamente, los pasos 1 y 4 se ejecutan solo una vez.

- El paso 3 se ejecuta como máximo m veces porque cada vez, excepto la última, marca un nodo adicional en G.

- Por lo tanto, el número total de pasos utilizadas es como máximo $1 + 1 + m$, lo que da un polinomio del tamaño de G.

- Los pasos 1 y 4 de M se implementan fácilmente en tiempo polinomial en cualquier modelo determinista razonable.

- El paso 3 implica un escaneo de la entrada y una prueba de si ciertos nodos están marcados, que también se implementa fácilmente en tiempo polinomial.

- Por tanto, M es un algoritmo en tiempo polinomial para $PATH$.

La clase NP I

- Como observamos en la sección 2, podemos evitar la búsqueda por fuerza bruta en muchos problemas y obtener soluciones de tiempo polinomial.

La clase NP I

- Sin embargo, los intentos de evitar la fuerza bruta en algunos otros problemas, incluidos muchos interesantes y útiles, no han tenido éxito y no se sabe que existan algoritmos de tiempo polinomial que los resuelvan.

La clase NP I

- ¿Por qué no hemos logrado encontrar algoritmos de tiempo polinomial para estos problemas?

La clase NP I

- No sabemos la respuesta a esta importante pregunta.

La clase NP I

- Quizás estos problemas tengan algoritmos de tiempo polinomial aún sin descubrir que se basan en principios desconocidos.

La clase NP II

- O posiblemente algunos de estos problemas simplemente **no se pueden resolver** en tiempo polinomial. Pueden ser intrínsecamente difíciles.

La clase NP II

- Un descubrimiento notable relacionado con esta cuestión muestra que las complejidades de muchos problemas están vinculadas.

La clase NP II

- Se puede utilizar un algoritmo de tiempo polinomial para uno de esos problemas para resolver toda una clase de problemas.

La clase NP II

- Para comprender este fenómeno, comencemos con un ejemplo.

Camino Hamiltoniano I

Un camino hamiltoniano en un grafo dirigido G es una ruta dirigida que atraviesa cada nodo exactamente una vez. Consideramos el problema de probar si un grafo dirigido contiene un camino hamiltoniano que conecta dos nodos específicos, como se muestra en la Figura 1.

Camino Hamiltoniano I

Un camino hamiltoniano en un grafo dirigido G es una ruta dirigida que atraviesa cada nodo exactamente una vez. Consideramos el problema de probar si un grafo dirigido contiene un camino hamiltoniano que conecta dos nodos específicos, como se muestra en la Figura 1.

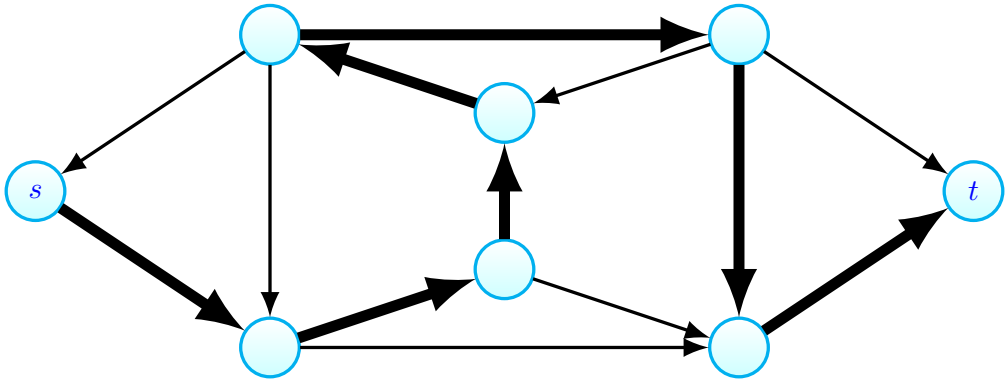

Figura 1: Un grafo con un camino Hamiltoniano representado con aristas gruesas (el camino recorre todos los nodos exactamente una vez, empezando en s y terminando en t).

Camino Hamiltoniano I

Sea

$HAMPATH = \{\langle G, s, t\rangle | G$ es un grafo dirigido con un camino Hamiltoniano de s a $t\}$.

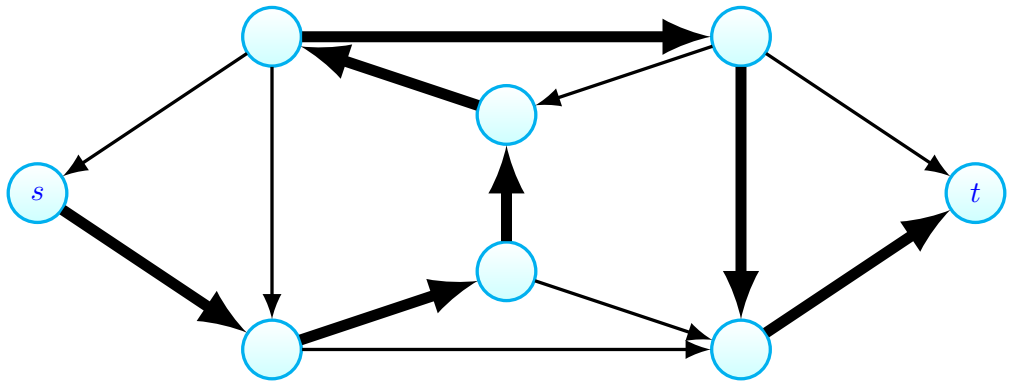

Figura 1: Un grafo con un camino Hamiltoniano representado con aristas gruesas (el camino recorre todos los nodos exactamente una vez, empezando en s y terminando en t).

Camino Hamiltoniano II

- Podemos obtener fácilmente un algoritmo de tiempo exponencial para el problema $HAMPATH$ modificando el algoritmo de fuerza bruta para $PATH$ dado en el Teorema 4.

Camino Hamiltoniano II

- Sólo necesitamos agregar una prueba para verificar que el camino potencial sea hamiltoniano. Nadie sabe si $HAMPATH$ se puede resolver en tiempo polinomial.

Camino Hamiltoniano II

- El problema $HAMPATH$ tiene una característica llamada **verificabilidad polinomial** que es importante para comprender su complejidad.

Camino Hamiltoniano III

- Aunque no conocemos una forma rápida (es decir, en tiempo polinomial) de determinar si un grafo contiene un camino hamiltoniano, si dicho camino se descubriera de alguna manera (tal vez usando el algoritmo en tiempo exponencial), podríamos convencer fácilmente a alguien más de su existencia simplemente presentándolo.

Camino Hamiltoniano III

- En otras palabras, verificar la existencia de un camino hamiltoniano puede ser mucho más fácil que determinar su existencia.

Camino Hamiltoniano IV

- Algunos problemas pueden no ser verificables polinomialmente.

Camino Hamiltoniano IV

- Por ejemplo, tome $\overline{HAMPATH}$, el complemento del problema $HAMPATH$.

Camino Hamiltoniano IV

- Incluso si pudiéramos determinar (de alguna manera) que un grafo no tiene un camino hamiltoniano.

Camino Hamiltoniano IV

- No conocemos una forma para que alguien más verifique su inexistencia sin usar el mismo algoritmo de tiempo exponencial para tomar la determinación en primer lugar.

Verificador I

Definición 5

Un verificador para un lenguaje A es un algoritmo V, donde

$$A = \{w | V \text{ acepta } \langle w, c \rangle \text{ para alguna cadena } c\}.$$

Medimos el tiempo de un verificador solo en términos de la longitud de w, por lo que un **verificador de tiempo polinomial** se ejecuta en tiempo polinomial en la longitud de w. Un lenguaje A es **verificable polinomialmente** si tiene un verificador de tiempo polinomial.

Verificador II

- Un verificador usa información adicional, representada por el símbolo c en la Definición 5, para verificar que una cadena w es miembro de A.

Verificador II

- Esta información se llama **certificado** o **prueba** de pertenencia a A.

Verificador II

- Observe que para los verificadores polinomiales, el certificado tiene una longitud polinomial (en la longitud de w) porque eso es todo lo que el verificador puede acceder en su límite de tiempo.

Verificador II

- Apliquemos esta definición a los lenguajes $HAMPATH$ y $COMPOSITES$.

Verificador III

- Para el problema $HAMPATH$, un certificado para una cadena $\langle G, s, t \rangle \in HAMPATH$ simplemente es una ruta hamiltoniana de s a t.

Verificador III

- Para el problema $COMPOSITES$, un certificado para el número compuesto x simplemente es uno de sus divisores.

Verificador III

- En ambos casos, el verificador puede comprobar en tiempo polinomial que la entrada está en el lenguaje partiendo del certificado que se ha dado.

La clase NP I

Definición 6

NP es la clase de lenguajes que tienen verificadores de tiempo polinomial.

La clase NP II

- La clase NP es importante porque contiene muchos problemas de interés práctico. De la discusión anterior, tanto $HAMPATH$ como $COMPOSITES$ son miembros de NP.

La clase NP II

- Como mencionamos, $COMPOSITES$ también es miembro de P, que es un subconjunto de NP; pero demostrar este resultado más fuerte es mucho más difícil.

La clase NP II

- El término NP proviene de **tiempo polinomial no determinista** y se deriva de una caracterización alternativa mediante el uso de máquinas de Turing de tiempo polinomial no deterministas.

La clase NP II

- Los problemas en NP a veces se denominan problemas NP.

La clase NP III

- Definimos el tiempo de una máquina no determinista como el tiempo utilizado por la rama de cálculo más larga.

La clase NP III

- La siguiente es una máquina de Turing no determinista (NTM) que decide el problema $HAMPATH$ en tiempo polinomial no determinista.

La clase NP IV

N_1 = "Como entrada recibe $\langle G, s, t \rangle$, donde G es un grafo dirigido con nodos s y t.

1. Escribe una lista de m números, $p_1, \ldots, p_m$, donde m es el número de nodos en G. Cada número en la lista se selecciona de manera no determinista para que esté entre 1 y m.
2. Revisa repeticiones en la lista. Si se encuentra alguna, *rechaza*.
3. Revisa si $s = p_1$ y $t = p_m$. Si no es así, *rechaza*.
4. Para cada i entre 1 y $m - 1$, revisa si (p_i, p_{i+1}) es una arista en G. Si alguna no lo es *rechaza*. En otro caso, todas las pruebas se han pasado, por lo tanto *acepta*."

La clase NP V

- Para analizar este algoritmo y verificar que se ejecuta en tiempo polinomial no determinista, examinamos cada uno de sus pasos.

La clase NP V

- En el paso 1, la selección no determinista se ejecuta claramente en tiempo polinomial.

La clase NP V

- En los pasos 2 y 3, cada parte es una verificación simple, por lo que juntas se ejecutan en tiempo polinomial.

La clase NP V

- Finalmente, el paso 4 también se ejecuta claramente en tiempo polinomial.

La clase NP V

- Por lo tanto, este algoritmo se ejecuta en un tiempo polinomial no determinista.

La clase NP VI

Teorema 7

Un lenguaje está en NP si y sólo si es decidido en tiempo polinomial por alguna máquina de Turing no determinista.

Teorema 7

Un lenguaje está en NP si y sólo si es decidido en tiempo polinomial por alguna máquina de Turing no determinista.

IDEA DE LA PRUEBA

- Mostramos cómo convertir un verificador de tiempo polinomial en una NTM de tiempo polinomial equivalente y viceversa.

Teorema 7

Un lenguaje está en NP si y sólo si es decidido en tiempo polinomial por alguna máquina de Turing no determinista.

IDEA DE LA PRUEBA

- La NTM simula al verificador adivinando el certificado.

Teorema 7

Un lenguaje está en NP si y sólo si es decidido en tiempo polinomial por alguna máquina de Turing no determinista.

IDEA DE LA PRUEBA

- El verificador simula la NTM utilizando como certificado la rama que acepta.

La clase NP VII

- Para la dirección hacia adelante de este teorema, sea $A \in$ NP y demuestre que A es decidido en tiempo polinomial por una NTM N.

La clase NP VII

- Sea V el verificador de tiempo polinomial para A que existe según la definición de NP. Suponga que V es una TM que se ejecuta en el tiempo n^k y construya N de la siguiente manera.

La clase NP VII

- Sea V el verificador de tiempo polinomial para A que existe según la definición de NP. Suponga que V es una TM que se ejecuta en el tiempo n^k y construya N de la siguiente manera.

$N =$ "Como entrada recibe w de longitud n:

1. Selecciona de manera no determinista una cadena c de longitud a lo más n^k.
2. Ejecuta V con entrada $\langle w, c \rangle$.
3. Si V acepta entonces *acepta*, de lo contrario *rechaza*."

La clase NP VIII

- Para probar la otra dirección del teorema, suponga que A se decide en tiempo polinomial por una NTM N y construya un verificador de tiempo polinomial V como sigue.

La clase NP VIII

- Para probar la otra dirección del teorema, suponga que A se decide en tiempo polinomial por una NTM N y construya un verificador de tiempo polinomial V como sigue.

$V =$ "Como entrada recibe $\langle w, c \rangle$, donde w y c son cadenas:

1. Simula N con la entrada w, tratando cada símbolo de c como una descripción de la elección no determinista a realizar en cada paso.
2. Si acepta esta rama del cálculo de N entonces *acepta*, de lo contrario *rechaza*."

La clase NP IX

Definimos la clase de complejidad temporal no determinista $\mathrm{NTIME}(t(n))$ de manera análoga a la clase de complejidad temporal determinista $\mathrm{TIME}(t(n))$.

La clase NP IX

Definición 8

Sea $t : \mathbb{N} \to \mathbb{R}^+$ una función. Defina la **clase de complejidad ntime**, en símbolos $\mathrm{NTIME}(t(n))$, como la colección de todos los lenguajes que son decidibles por una máquina de Turing no determinista en tiempo $O(t(n))$.

La clase NP IX

Definición 8

Sea $t : \mathbb{N} \to \mathbb{R}^+$ una función. Defina la **clase de complejidad ntime**, en símbolos $\mathrm{NTIME}(t(n))$, como la colección de todos los lenguajes que son decidibles por una máquina de Turing no determinista en tiempo $O(t(n))$.

Corolario 9

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k).$$

La clase NP X

- La clase NP es insensible a la elección de un modelo computacional no determinista razonable porque todos esos modelos son polinomialmente equivalentes.

La clase NP X

- Al describir y analizar algoritmos de tiempo polinomial no determinista, seguimos las convenciones anteriores para algoritmos de tiempo polinomial determinista.

La clase NP X

- Cada etapa de un algoritmo de tiempo polinomial no determinista debe tener una implementación obvia en tiempo polinomial no determinista en un modelo computacional no determinista razonable.

La clase NP X

- Analizamos el algoritmo para mostrar que cada rama usa, como mucho, polinomialmente muchas etapas.

Clique I

Un **clique** en un grafo no dirigido es un subgrafo en el que cada dos nodos están conectados por una arista. Una clique$-k$ es un clique que contiene k nodos. La Figura 2 ilustra un grafo con un clique-5.

Clique I

Un **clique** en un grafo no dirigido es un subgrafo en el que cada dos nodos están conectados por una arista. Una clique$-k$ es un clique que contiene k nodos. La Figura 2 ilustra un grafo con un clique-5.

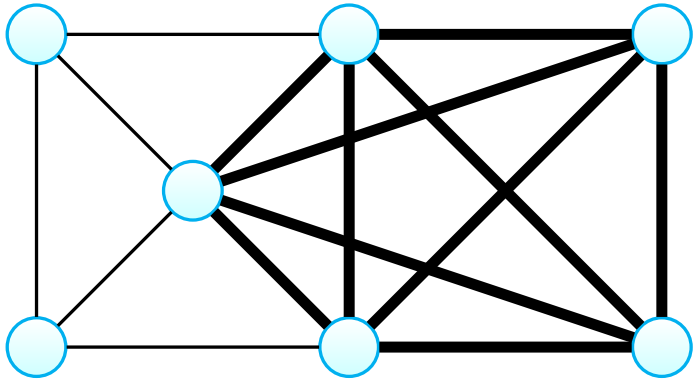

Figura 2: Un grafo con un clique-5.

Clique I

El problema $CLIQUE$ es determinar si un grafo contiene un clique de un tamaño específico. Sea

$$CLIQUE = \{\langle G, k\rangle | G \text{ es un grafo no dirigido con un clique}-k\}.$$

Clique II

Teorema 10

$CLIQUE$ está en NP.

Clique II

Teorema 10

$CLIQUE$ está en NP.

IDEA DE LA PRUEBA El clique es el certificado.

Clique II

Teorema 10

$CLIQUE$ está en NP.

PRUEBA El siguiente es un verificador V para $CLIQUE$.
$V =$ "Como entrada recibe $\langle\langle G, k\rangle, c\rangle$:

1. Prueba si c es un subgrafo con k nodos en G.
2. Prueba si G contiene todas las aristas que conectan los nodos en c.
3. Si ambas pruebas tienen éxito, entonces *acepta*; de lo contrario *rechaza*."

Clique III

PRUEBA ALTERNATIVA Si prefiere pensar NP en términos de máquinas de Turing de tiempo polinomial no determinista, puede probar este teorema dando una que decida $CLIQUE$.

Clique III

PRUEBA ALTERNATIVA Si prefiere pensar NP en términos de máquinas de Turing de tiempo polinomial no determinista, puede probar este teorema dando una que decida $CLIQUE$.
$N =$ "Como entrada recibe $\langle G, k \rangle$, donde G es un grafo:

1. De manera no determinista selecciona un subconjunto c de k nodos de G.
2. Prueba si G contiene todas las aristas que conectan nodos en c.
3. Si es así, *acepta*; de lo contrario, *rechaza*."

Clique IV

Observe la similitud entre las dos pruebas.

Clique IV

$V =$ "Como entrada recibe $\langle\langle G, k\rangle, c\rangle$:

1. Prueba si c es un subgrafo con k nodos en G.
2. Prueba si G contiene todas las aristas que conectan los nodos en c.
3. Si ambas pruebas tienen éxito, entonces *acepta*; de lo contrario *rechaza*."

$N =$ "Como entrada recibe $\langle G, k\rangle$, donde G es un grafo:

1. De manera no determinista selecciona un subconjunto c de k nodos de G.
2. Prueba si G contiene todas las aristas que conectan nodos en c.
3. Si es así, *acepta*; de lo contrario, *rechaza*."

Observación 11

- Observe que los complementos de estos conjuntos, $\overline{CLIQUE}$ y $\overline{SUBSET\text{–}SUM}$, no son evidentemente miembros de NP. Verificar que algo **no** está presente parece ser más difícil que verificar que **sí** está presente.

Observación 11

- Creamos una clase de complejidad separada, llamada coNP, que contiene los lenguajes que son complementos de los lenguajes en NP.

Observación 11

- No sabemos si coNP es diferente de NP.

¿P = NP? I

- Como hemos dicho, NP es la clase de lenguajes que se pueden resolver en tiempo polinomial en una máquina de Turing no determinista.

¿P = NP? I

- De forma equivalente, es la clase de lenguajes mediante los cuales se puede verificar la pertenencia al lenguaje en tiempo polinomial.

¿P = NP? I

- P es la clase de lenguajes donde se puede decidir la pertenencia en tiempo polinomial.

¿P = NP? I

- Resumimos esta información de la siguiente manera, donde nos referimos vagamente al tiempo polinomial resoluble como resoluble rápidamente".

¿P = NP? I

- Resumimos esta información de la siguiente manera, donde nos referimos vagamente al tiempo polinomial resoluble como resoluble rápidamente".
  - P = la clase de lenguajes para los que se puede **decidir** rápidamente la membresía.

¿P = NP? I

- Resumimos esta información de la siguiente manera, donde nos referimos vagamente al tiempo polinomial resoluble como resoluble rápidamente".
  - P = la clase de lenguajes para los que se puede **decidir** rápidamente la membresía.
  - NP = la clase de lenguajes para los que se puede **verificar** rápidamente la membresía.

¿P = NP? II

- Hemos presentado ejemplos de lenguajes, como $HAMPATH$ y $CLIQUE$, que son miembros de NP pero que no se sabe si están en P.

¿P = NP? II

- El poder de la verificabilidad polinomial parece ser mucho mayor que el de la decidibilidad polinomial.

¿P = NP? II

- Pero, por difícil que sea de imaginar, P y NP podrían ser iguales.

¿P = NP? II

- No se ha podido **demostrar** la existencia de un solo lenguaje en NP que esté en P.

¿P = NP? II

- La pregunata si P = NP es uno de los mayores problemas sin resolver en la computación teórica y las matemáticas contemporáneas.

¿P = NP? II

- Si estas clases fueran iguales, cualquier problema polinomialmente verificable sería polinomialmente decidible.

¿P = NP? III

- La mayoría de los investigadores creen que las dos clases no son iguales porque la gente ha invertido un esfuerzo enorme para encontrar algoritmos de tiempo polinomial para ciertos problemas en NP, sin éxito.

¿P = NP? III

- Los investigadores también han intentado demostrar que las clases son desiguales, pero eso implicaría demostrar que no existe un algoritmo rápido para reemplazar la búsqueda por fuerza bruta.

¿P = NP? III

- Hacerlo está actualmente más allá del alcance científico.

¿P = NP? IV

La Figura 3 muestra las dos posibilidades.

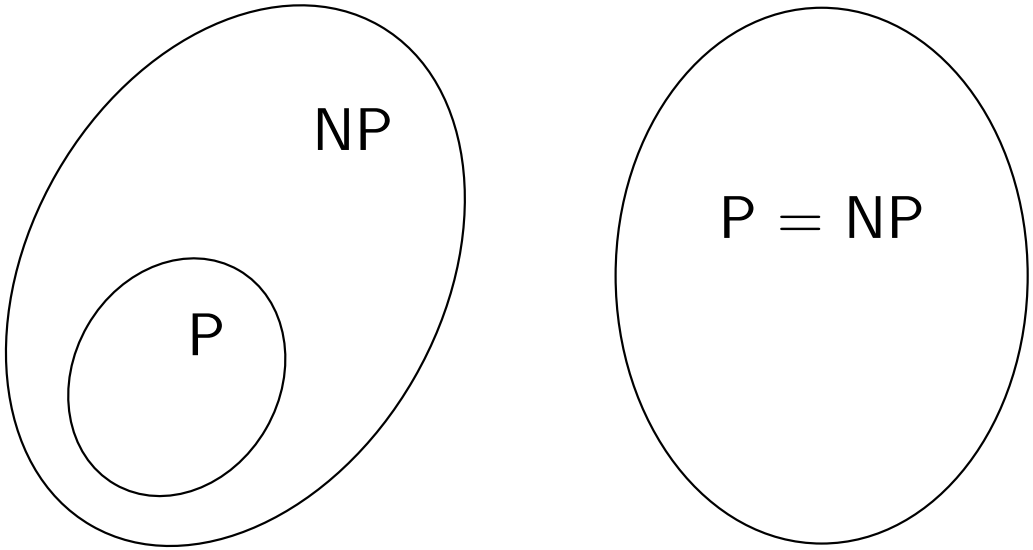

Figura 3: Una de las dos posibilidades es correcta.

¿P = NP? V

El mejor método determinista actualmente conocido para decidir lenguajes en NP utiliza tiempo exponencial.

¿P = NP? V

En otras palabras, podemos demostrar que

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k}),$$

¿P = NP? V

En otras palabras, podemos demostrar que

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k}),$$

pero no sabemos si NP está contenido en una clase de complejidad temporal determinista más pequeña.

Completitud-NP I

- Un avance importante sobre la pregunta P versus NP se produjo a principios de la década de 1970 con el trabajo de Stephen Cook y Leonid Levin.

Completitud-NP I

- Descubrieron ciertos problemas en NP cuya complejidad individual está relacionada con la de toda la clase.

Completitud-NP I

- Si existe un algoritmo de tiempo polinomial para cualquiera de estos problemas, todos los problemas en NP podrían resolverse en tiempo polinomial.

Completitud-NP I

- Estos problemas se denominan NP-completos. El fenómeno de la completitud-NP es importante tanto por razones teóricas como prácticas.

Completitud-NP II

- En el aspecto teórico, un investigador que intente demostrar que P no es igual a NP puede centrarse en un problema NP completo.

Completitud-NP II

- Si algún problema en NP requiere más que tiempo polinomial, uno NP-completo también lo requerirá.

Completitud-NP II

- Además, un investigador que intente demostrar que P es igual a NP sólo necesita encontrar un algoritmo de tiempo polinomial para un problema NP-completo para lograr este objetivo.

Completitud-NP III

- Desde el punto de vista práctico, el fenómeno de la completitud-NP puede evitar perder tiempo buscando un algoritmo de tiempo polinomial inexistente para resolver un problema particular.

Completitud-NP III

- Aunque no tengamos las matemáticas necesarias para demostrar que el problema no tiene solución en tiempo polinomial, creemos que P no es igual a NP.

Completitud-NP III

- Entonces, demostrar que un problema es NP-completo es una fuerte evidencia de su **no polinomialidad**.

- El primer problema NP-completo que presentamos se llama **problema de satisfacibilidad**.

- Recuerde que las variables que pueden tomar los valores TRUE y FALSE se denominan **variables booleanas**.

- Usualmente, representamos TRUE por 1 y FALSE por 0. Las **operaciones booleanas** AND, OR y NOT, representadas por los símbolos $\wedge, \vee$, y $\neg$ respectivamente, se describen en la siguiente lista.

- Usamos la barra superior como forma abreviada del símbolo $\neg$, por lo que $\overline{x}$ significa $\neg x$.

$$0 \wedge 0 = 0 \qquad 0 \vee 0 = 0 \qquad \overline{0} = 1$$
$$0 \wedge 1 = 0 \qquad 0 \vee 1 = 1 \qquad \overline{1} = 0$$
$$1 \wedge 0 = 0 \qquad 1 \vee 0 = 1$$
$$1 \wedge 1 = 1 \qquad 1 \vee 1 = 1$$

- Una **fórmula booleana** es una expresión que involucra variables y operaciones booleanas. Por ejemplo,

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

es una fórmula booleana.

- Una fórmula booleana es **satisfactoria** si alguna asignación de 0s y 1s a sus variables hace que la fórmula se evalúe a 1.

- Una **fórmula booleana** es una expresión que involucra variables y operaciones booleanas. Por ejemplo,

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

es una fórmula booleana.

- La fórmula anterior es satisfactoria porque la asignación $x = 0$, $y = 1$ y $z = 0$ hace que ϕ se evalúe a 1. Decimos la asignción satisface ϕ.

- El problema de satisfacibilidad es probar si una fórmula booleana es satisfactoria. Sea

$$SAT = \{\langle\phi\rangle | \phi \text{ es una fórmula booleana satisfactoria}\}.$$

Ahora planteamos un teorema que vincula la complejidad del problema SAT con las complejidades de todos los problemas en NP.

Teorema 12

$SAT \in$ P si y sólo si P $=$ NP.

A continuación, desarrollamos el método que es fundamental para la demostración de este teorema.

Reducibilidad en tiempo polinomial I

- Cuando el problema A se reduce al problema B, se puede usar una solución de B para resolver A.

Reducibilidad en tiempo polinomial I

- Ahora definimos una versión de reducibilidad que toma en cuenta la eficiencia del cálculo.

Reducibilidad en tiempo polinomial I

- Cuando el problema A se puede reducir eficientemente al problema B, se puede usar una solución eficiente para B para resolver A de manera eficiente.

Reducibilidad en tiempo polinomial II

Definición 13

Una función $f : \Sigma^* \to \Sigma^*$ es una **función computable en tiempo polinomial** si existe alguna máquina de Turing M de tiempo polinomial que se detiene dejando $f(w)$ en su cinta, cuando se inicia con cualquier entrada w.

Reducibilidad en tiempo polinomial II

Definición 14

El lenguaje A es **reducible mediante un mapeo en tiempo polinomial**, o simplemente **reducible en tiempo polinomial**, al lenguaje B, en símbolos $A \leq_P B$, si existe una función computable en tiempo polinomial $f : \Sigma^* \to \Sigma^*$, donde para cada w,

$$w \in A \Leftrightarrow f(w) \in B.$$

A la función f le llamamos **reducción en tiempo polinomial** de A a B.

Reducibilidad en tiempo polinomial III

- Hay otras formas eficientes de reducibilidad disponibles, pero la reducibilidad en tiempo polinomial es una forma simple que es adecuada para nuestros propósitos, por lo que no discutiremos las otras aquí.

Reducibilidad en tiempo polinomial III

- Al igual que con un mapeo de reducción ordinario, una reducción en tiempo polinomial de A a B proporciona una forma de convertir las pruebas de pertenencia en A a pruebas de pertenencia en B, pero ahora la conversión se realiza de manera eficiente.

Reducibilidad en tiempo polinomial IV

- Para probar si $w \in A$, usamos la reducción f para mapear w a $f(w)$ y probar si $f(w) \in B$.

Reducibilidad en tiempo polinomial IV

- Si un lenguaje es reducible en tiempo polinomial a un lenguaje que ya se sabe que tiene una solución en tiempo polinomial, obtenemos una solución en tiempo polinomial para el lenguaje original, como en el siguiente teorema.

Reducibilidad en tiempo polinomial V

Teorema 15

Si $A \leq_P B$ y $B \in$ P entonces $A \in P$.

Reducibilidad en tiempo polinomial V

Teorema 15

Si $A \leq_P B$ y $B \in$ P entonces $A \in P$.

PRUEBA Sea M el algoritmo en tiempo polinomial que decide B y f la reducción en tiempo polinomial de A a B. Describimos un algoritmo de tiempo polinomial N que decide A de la siguiente manera.

Reducibilidad en tiempo polinomial V

Teorema 15

Si $A \leq_P B$ y $B \in$ P entonces $A \in P$.

PRUEBA Sea M el algoritmo en tiempo polinomial que decide B y f la reducción en tiempo polinomial de A a B. Describimos un algoritmo de tiempo polinomial N que decide A de la siguiente manera.
$N =$ "Como entrada recibe w:

1. Calcula $f(w)$.
2. Ejecuta M con $f(w)$ sobre la cinta y responde lo que M responda.

Reducibilidad en tiempo polinomial VI

- Tenemos que $w \in A$ siempre que $f(w) \in B$ porque f es una reducción de A a B.

Reducibilidad en tiempo polinomial VI

- Por lo tanto, M acepta $f(w)$ siempre que $w \in A$. Además, N se ejecuta en tiempo polinomial porque cada un de sus dos pasos se ejecuta en tiempo polinomial.

Reducibilidad en tiempo polinomial VI

- Tenga en cuenta que el paso 2 se ejecuta en tiempo polinomial porque la composición de dos polinomios es un polinomio.

$3SAT$ I

- Antes de demostrar una reducción en tiempo polinomial, presentamos $3SAT$, un caso especial del problema de satisfacibilidad en el que todas las fórmulas están en una forma especial.

- Una **literal** es una variable booleana o una variable booleana negada, como en x o $\overline{x}$.

- Una **cláusula** son varias literales conectados con $\vee$s, como en $(x_1 \vee x_2 \vee x_3 \vee x_4)$. Una fórmula booleana está en **forma normal conjuntiva**, llamada **fórmula-cnf**, si comprende varias cláusulas conectadas con $\wedge$s, como en

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \vee (x_3 \vee \overline{x_6}).$$

- Es una **fórmula-3cnf** si todas las cláusulas tienen tres literales, como en

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

- Sea

$$3SAT = \{\langle\phi\rangle | \phi \text{ es una fórmula-3cnf satisfactoria}\}.$$

- Si una asignación satisface una fórmula-cnf, cada cláusula debe contener al menos una literal que se evalúe a 1.

- El siguiente teorema presenta una reducción en tiempo polinomial del problema $3SAT$ al problema $CLIQUE$.

Teorema 16

$3SAT$ es reducible en tiempo polinomial a $CLIQUE$.

Teorema 16

$3SAT$ es reducible en tiempo polinomial a $CLIQUE$.

IDEA DE LA PRUEBA

Teorema 16

$3SAT$ es reducible en tiempo polinomial a $CLIQUE$.

IDEA DE LA PRUEBA

- La reducción en tiempo polinomial f que mostramos de $3SAT$ a $CLIQUE$ convierte fórmulas en grafos.

Teorema 16

$3SAT$ es reducible en tiempo polinomial a $CLIQUE$.

IDEA DE LA PRUEBA

- En los grafos construidos, los cliques de un tamaño específico corresponden a asignaciones satisfactorias de la fórmula.

Teorema 16

$3SAT$ es reducible en tiempo polinomial a $CLIQUE$.

IDEA DE LA PRUEBA

- Las estructuras dentro del grafo están diseñadas para imitar el comportamiento de las variables y cláusulas.

PRUEBA Sea ϕ una fórmula con k cláusulas como

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$$

PRUEBA Sea ϕ una fórmula con k cláusulas como

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$$

- La reducción f genera la cadena $\langle G, k \rangle$, donde G es un grafo no dirigido definido como sigue.

PRUEBA Sea ϕ una fórmula con k cláusulas como

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$$

- Los nodos de G están organizados en k grupos de tres nodos, los grupos se llama triples, $t_1, \ldots, t_k$.

PRUEBA Sea ϕ una fórmula con k cláusulas como

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$$

- Cada triple corresponde a una de las cláusulas en ϕ, y cada nodo en un triple corresponde a una literal en la cláusula asociada.

PRUEBA Sea ϕ una fórmula con k cláusulas como

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$$

- Etiquete cada nodo de G con su literal correspondiente en ϕ.

- Las aristas de G conectan todos menos dos tipos de pares de nodos en G.

- No hay una arista presente entre nodos en el mismo triple, y no hay una arista presente entre dos nodos con etiquetas contradictorias, como en x_2 y $\overline{x_2}$.

- No hay una arista presente entre nodos en el mismo triple, y no hay una arista presente entre dos nodos con etiquetas contradictorias, como en x_2 y $\overline{x_2}$.
- La Figura 4 ilustra esta construcción cuando $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

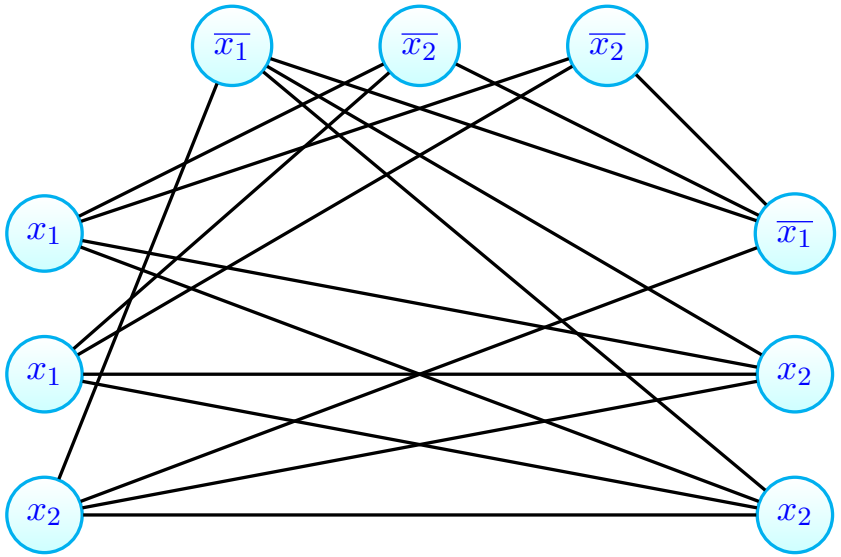

Figura 4: El grafo que produce la reducción para $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

- Ahora demostramos por qué funciona esta construcción.

- Demostramos que ϕ es satisfactoria si G tiene un clique$-k$.

- Suponga que ϕ tiene una asignación satisfactoria.

- En esa asignación satisfactoria, al menos una literal es verdadera en cada cláusula.

- En cada triple de G, seleccionamos un nodo correspondiente a una literal verdadera en la asignación satisfactoria.

- Si más de una literal es verdadera en una cláusula en particular, elegimos arbitrariamente una de las literales verdaderas.

- Los nodos recién seleccionados forman un clique$-k$.

- El número de nodos seleccionados es k porque elegimos uno para cada uno de los k triples.

- Cada par de nodos seleccionados está unido por una arista porque ningún par se ajusta a una de las excepciones descritas anteriormente.

- No podían ser del mismo triple porque seleccionamos sólo un nodo por triple.

- No podían tener etiquetas contradictorias porque las literales asociados eran ambas verdaderas en la asignación satisfactoria.

- Por tanto, G contiene un clique$-k$.

- Suponga que G tiene un clique$-k$. No hay dos nodos del clique en el mismo triple porque los nodos del mismo triple no están conectados por aristas.

- Por lo tanto, cada uno de los k triples contiene exactamente uno de los k nodos clique.

- Asignamos valores de verdad a las variables de ϕ para que cada literal que etiquete un nodo clique se convierta en verdadero.

- Hacerlo siempre es posible porque dos nodos etiquetados de manera contradictoria no están conectados por una arista y, por lo tanto, ambos no pueden estar en el clique.

- Esta asignación a las variables satisface ϕ porque cada triple contiene un nodo clique y, por lo tanto, cada cláusula contiene una literal a la que se le asigna TRUE.

- Por tanto, ϕ es satisfactoria.

- Los teoremas 15 y 16 nos dicen que si $CLIQUE$ se puede resolver en tiempo polinomial, también $3SAT$ se puede resolver en tiempo polinomial.

- A primera vista, esta conexión entre estos dos problemas parece bastante notable porque, superficialmente, son bastante diferentes.

- Pero la reducibilidad de tiempo polinomial nos permite vincular sus complejidades.

- Ahora pasamos a una definición que nos permitirá vincular de manera similar las complejidades de toda una clase de problemas.

Completitud-NP I

Definición 17

Un lenguaje B es **NP-completo** si satisface dos condiciones:

1. B está en NP y
2. todo A en NP es reducible en tiempo polinomial a B.

Completitud-NP I

Definición 17

Un lenguaje B es **NP-completo** si satisface dos condiciones:

1. B está en NP y
2. todo A en NP es reducible en tiempo polinomial a B.

Teorema 18

Si B es NP-completo y $B \in$ P, entonces P $=$ NP.

Completitud-NP I

Definición 17

Un lenguaje B es **NP-completo** si satisface dos condiciones:

1. B está en NP y
2. todo A en NP es reducible en tiempo polinomial a B.

Teorema 18

Si B es NP-completo y $B \in$ P, entonces P $=$ NP.

PRUEBA Este teorema se deriva directamente de la definición de reducibilidad en tiempo polinomial.

Completitud-NP II

Teorema 19

Si B es NP-completo y $B \leq_P C$ para C in NP, entonces C es NP-completo.

Completitud-NP II

Teorema 19

Si B es NP-completo y $B \leq_P C$ para C in NP, entonces C es NP-completo.

- **PRUEBA** Ya sabemos que C está en NP, por lo que debemos demostrar que cada A en NP es reducible en tiempo polinomial a C.

Completitud-NP II

Teorema 19

Si B es NP-completo y $B \leq_P C$ para C in NP, entonces C es NP-completo.

- Como B es NP-completo, cada lenguaje en NP es reducible en tiempo polinomial a B, y B a su vez es reducible en tiempo polinomial a C.

Completitud-NP II

Teorema 19

Si B es NP-completo y $B \leq_P C$ para C in NP, entonces C es NP-completo.

- Componga las reducciones de tiempo de polinomial; es decir, si A es reducible en tiempo polinomial a B y B es reducible en tiempo polinomial a C, entonces A es reducible en tiempo polinomial a C.

Completitud-NP II

Teorema 19

Si B es NP-completo y $B \leq_P C$ para C in NP, entonces C es NP-completo.

- Por tanto, cada lenguaje en NP es reducible en tiempo polinomial a C.

Completitud-NP III

- Una vez que tenemos un problema NP-completo, podemos obtener otros mediante reducciones de tiempo polinomial.

Completitud-NP III

- Sin embargo, establecer el primer problema NP-completo es más difícil.

Completitud-NP III

- Ahora lo hacemos probando que SAT es NP-completo.

Teorema 20

SAT es NP-completo.

Completitud-NP III

- Ahora lo hacemos probando que SAT es NP-completo.

Teorema 20

SAT es NP-completo.

- Este teorema implica el Teorema 12.

IDEA DE LA PRUEBA

- Demostrar que SAT está en NP es fácil y lo haremos en breve.

IDEA DE LA PRUEBA

- La parte difícil de la demostración es mostrar que cualquier lenguaje en NP es reducible en tiempo polinomial a SAT.

IDEA DE LA PRUEBA

- Para hacerlo, construimos una reducción en tiempo polinomial para cada lenguaje A en NP a SAT.

IDEA DE LA PRUEBA

- La reducción para A toma una cadena w y produce una fórmula booleana ϕ que simula la máquina NP para A con la entrada w.

IDEA DE LA PRUEBA

- Si la máquina acepta, ϕ tiene una asignación satisfactoria que corresponde al cálculo de aceptación.

IDEA DE LA PRUEBA

- Si la máquina no acepta, ninguna asignación satisface ϕ.

IDEA DE LA PRUEBA

- Por lo tanto, w está en A si y solo si ϕ es satisfactoria.

Completitud-NP V

- En realidad, construir la reducción para que funcione de esta manera es una tarea conceptualmente simple, aunque debemos lidiar con muchos detalles.

Completitud-NP V

- Una fórmula booleana puede contener las operaciones booleanas AND, OR y NOT, y estas operaciones forman la base de los circuitos utilizados en las computadoras electrónicas.

Completitud-NP V

- Por tanto, no sorprende el hecho de que podamos diseñar una fórmula booleana para simular una máquina de Turing.

Completitud-NP V

- Los detalles están en la implementación de esta idea.

Fin del curso

Muchas gracias