Análisis de Algoritmos II

Jesús Alexander Aranda Ph.D Robinson Duque, Ph.D
Juan Francisco Díaz, Ph. D

Universidad del Valle

jesus.aranda@correounivalle.edu.co
robinson.duque@correounivalle.edu.co
juanfco.diaz@correounivalle.edu.co

Programa de Ingeniería de Sistemas
Escuela de Ingeniería de Sistemas y Computación

Overview

Complejidad Computacional

La complejidad computacional está relacionada con el estudio de algoritmos.

Algoritmo

Es un procedimiento computacional bien definido que toma algún valor, o conjunto de valores, como entrada y produce algún valor, o conjunto de valores como salida.
Un algoritmo es entonces una secuencia de pasos computacionales que transforman una entrada en una salida.

Complejidad Computacional

Ejemplo: Ordenamiento
Entrada: Una secuencia de n números $a = (a_1, a_2, ..., a_n)$
Salida: Una permutación (reordenamiento) de a: $a' = (a'_1, a'_2, ..., a'_n)$ tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$

Se conoce como instancia del problema a una entrada específica:
Entrada (instancia): (31,41,59,26,41,58)
Salida: (26,31,41,41,58,59)

Complejidad Computacional

La complejidad computacional considera los algoritmos para resolver un problema dado. Existen diversas formas de medir la complejidad de un algoritmo. La complejidad se mide en función del tamaño de la entrada.

Clases de complejidad

- **Temporal:** tiempo requerido por un algoritmo para resolver un problema dado.
- **Espacial:** espacio que requieren las estructuras de datos de un algoritmo para resolver un problema dado.

Complejidad Computacional

¿Puedo superar las ineficiencias de algún algoritmo con una mejor máquina?

- El algoritmo `insertion sort`, toma en general $c_1 * n^2$ para ordenar n items
- El algoritmo `merge sort`, toma en general $c_2 * n * log_2 n$ para ordenar n items

Complejidad Computacional

Suponga que tenemos dos computadores A y B con capacidad para 10^{10} y 10^7 operaciones por segundo, respectivamente. Ejecutamos `insertion sort` en A y `merge sort` en B para ordenar 10 millones de números:

- Insertion sort (asuma $c_1 = 2$),
  $\frac{2*(10^7)^2 instrucciones}{10^{10} instrucciones/segundo} = 20{,}000 segundos$ (más de 5.5 horas)
- Merge sort (asuma $c_2 = 50$),
  $\frac{50*(10^7)*log_2(10^7) instrucciones}{10^7 instrucciones/segundo} = 1163 segundos$ (menos de 20 min)

La diferencia es más pronunciada para n más grandes. e.g., 100 millones de números (`insertion sort`: 23 días, `merge sort` toma menos de 4 horas). En conclusión, vale más un buen algoritmo que una buena máquina.

Complejidad Computacional

Suponga que tenemos dos computadores A y B con capacidad para 10^{10} y 10^7 operaciones por segundo, respectivamente. Ejecutamos `insertion sort` en A y `merge sort` en B para ordenar 10 millones de números:

- Insertion sort (asuma $c_1 = 2$),
  $\frac{2*(10^7)^2 instrucciones}{10^{10} instrucciones/segundo} = 20{,}000 segundos$ (más de 5.5 horas)
- Merge sort (asuma $c_2 = 50$),
  $\frac{50*(10^7)*log_2(10^7) instrucciones}{10^7 instrucciones/segundo} = 1163 segundos$ (menos de 20 min)

La diferencia es más pronunciada para n más grandes. e.g., 100 millones de números (`insertion sort`: 23 días, `merge sort` toma menos de 4 horas). En conclusión, vale más un buen algoritmo que una buena máquina.

Complejidad Computacional

Suponga que tenemos dos computadores A y B con capacidad para 10^{10} y 10^7 operaciones por segundo, respectivamente. Ejecutamos `insertion sort` en A y `merge sort` en B para ordenar 10 millones de números:

- Insertion sort (asuma $c_1 = 2$),
  $\frac{2*(10^7)^2 instrucciones}{10^{10} instrucciones/segundo} = 20{,}000 segundos$ (más de 5.5 horas)
- Merge sort (asuma $c_2 = 50$),
  $\frac{50*(10^7)*log_2(10^7) instrucciones}{10^7 instrucciones/segundo} = 1163 segundos$ (menos de 20 min)

La diferencia es más pronunciada para n más grandes. e.g., 100 millones de números (`insertion sort`: 23 días, `merge sort` toma menos de 4 horas). En conclusión, vale más un buen algoritmo que una buena máquina.

Complejidad Computacional

- Una de las principales características que se estudia en ciencias de la computación es el tiempo de ejecución de un algoritmo en función del tamaño de la entrada $T(n)$.
- La complejidad de un problema se asocia a la complejidad del mejor algoritmo para resolverlo.
- El tiempo de ejecución es expresado en notación asintótica:
  - $O(f(n))$
  - $\Theta(f(n))$
  - $\Omega(f(n))$
- Los problemas tienen diversos niveles de dificultad asociados al tiempo de ejecución, por ejemplo: $1 < log_2 n < \sqrt{n} < n < nlog_2 n < n^2 < n^3 < ... < 2^n < 3^n < ... < n! < n^n$

Complejidad Computacional

- Una de las principales características que se estudia en ciencias de la computación es el tiempo de ejecución de un algoritmo en función del tamaño de la entrada $T(n)$.
- La complejidad de un problema se asocia a la complejidad del mejor algoritmo para resolverlo.
- El tiempo de ejecución es expresado en notación asintótica:
  - $O(f(n))$
  - $\Theta(f(n))$
  - $\Omega(f(n))$
- Los problemas tienen diversos niveles de dificultad asociados al tiempo de ejecución, por ejemplo: $1 < log_2 n < \sqrt{n} < n < nlog_2 n < n^2 < n^3 < ... < 2^n < 3^n < ... < n! < n^n$

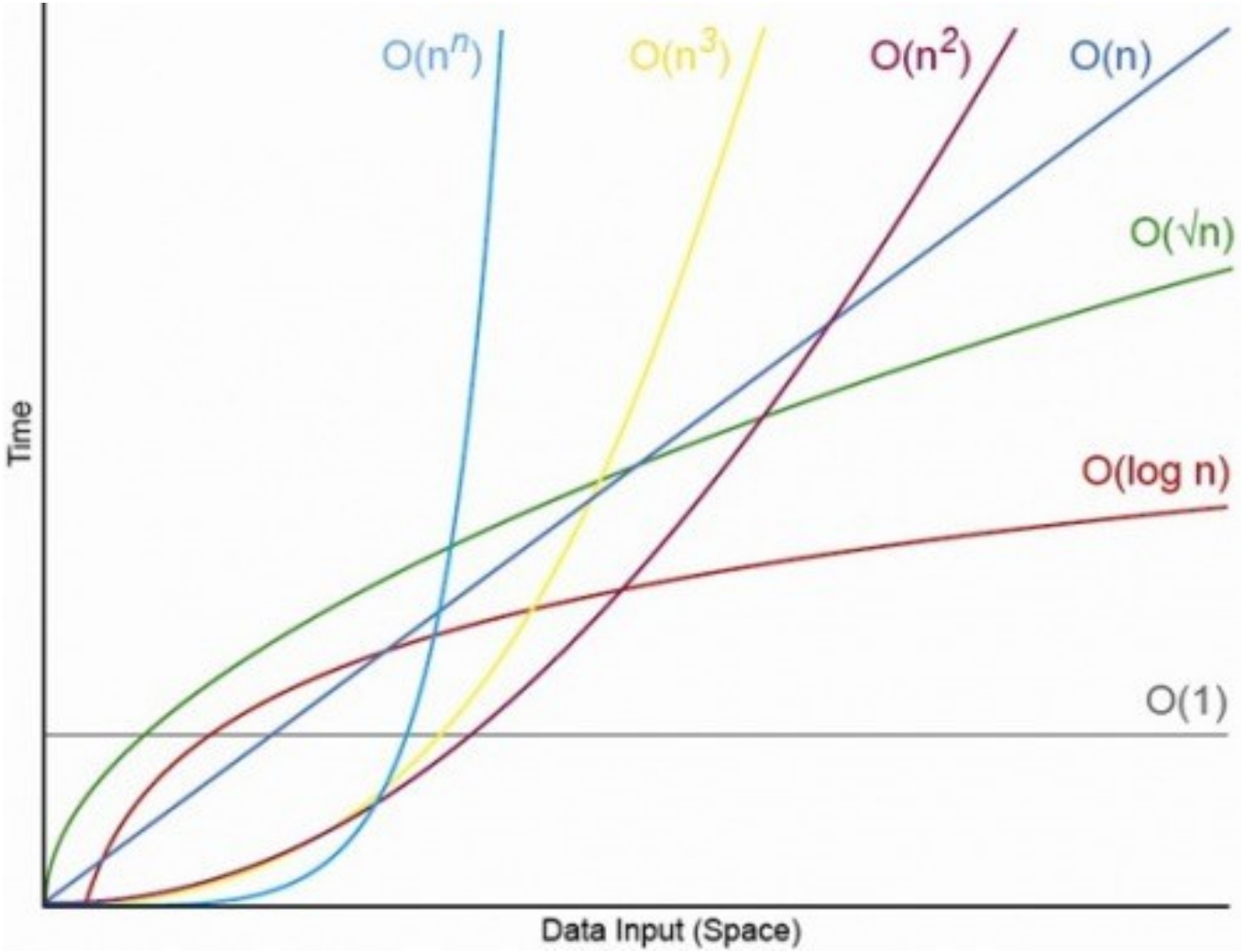
Complejidad Computacional

- Una de las principales características que se estudia en ciencias de la computación es el tiempo de ejecución de un algoritmo en función del tamaño de la entrada $T(n)$.
- La complejidad de un problema se asocia a la complejidad del mejor algoritmo para resolverlo.
- El tiempo de ejecución es expresado en notación asintótica:
  - $O(f(n))$
  - $\Theta(f(n))$
  - $\Omega(f(n))$
- Los problemas tienen diversos niveles de dificultad asociados al tiempo de ejecución, por ejemplo: $1 < log_2 n < \sqrt{n} < n < nlog_2 n < n^2 < n^3 < ... < 2^n < 3^n < ... < n! < n^n$

Complejidad Computacional

- Una de las principales características que se estudia en ciencias de la computación es el tiempo de ejecución de un algoritmo en función del tamaño de la entrada $T(n)$.
- La complejidad de un problema se asocia a la complejidad del mejor algoritmo para resolverlo.
- El tiempo de ejecución es expresado en notación asintótica:
  - $O(f(n))$
  - $\Theta(f(n))$
  - $\Omega(f(n))$
- Los problemas tienen diversos niveles de dificultad asociados al tiempo de ejecución, por ejemplo: $1 < log_2 n < \sqrt{n} < n < nlog_2 n < n^2 < n^3 < ... < 2^n < 3^n < ... < n! < n^n$

Complejidad Computacional



Complejidad Computacional

Ejemplos

- **Algoritmo de ordenamiento:** Ordenar n elementos tiene una complejidad temporal de $(nlog_2 n)$, cuando se utilizan *comparaciones* y si la complejidad de copiar y comparar dos elementos cualquiera es O(1).
- **Buscar elementos** en un árbol binario con n elementos tiene una complejidad en tiempo de $O(log_2 n)$, donde $log_2 n$ está asociado a la altura del árbol.

Problemas Decidibles e Indecidibles

Es necesario establecer un criterio que permita diferenciar y categorizar cualquier problema de acuerdo con su nivel de dificultad:

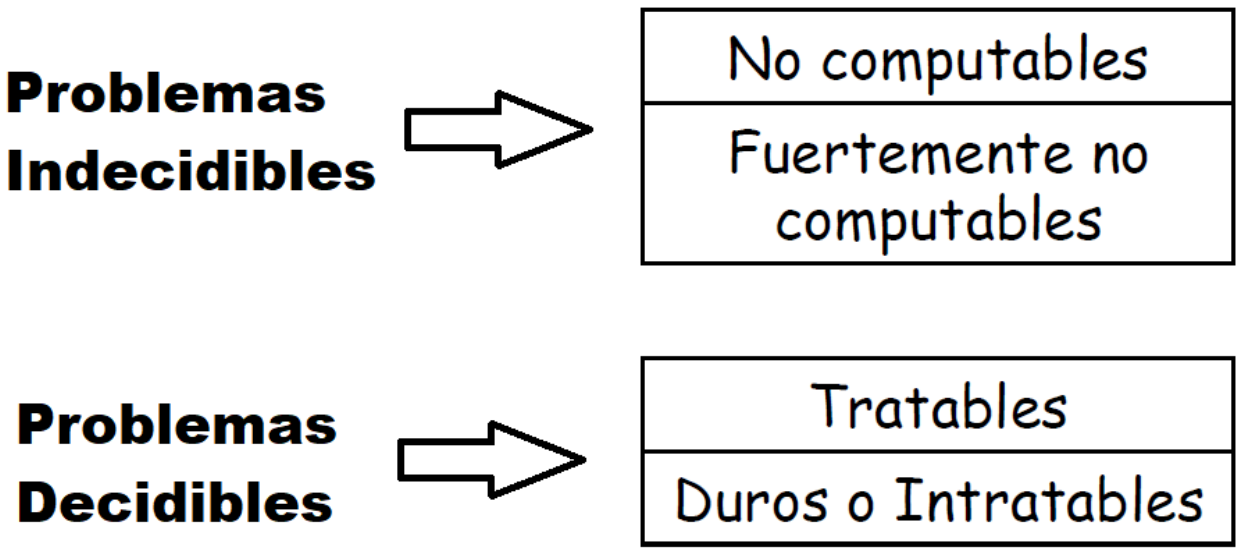
Problemas decidibles

Aquellos para los que existe un algoritmo para su cómputo, es decir, una MT que se detiene cuando se acepta la entrada o cuando no.

Problemas indecidibles

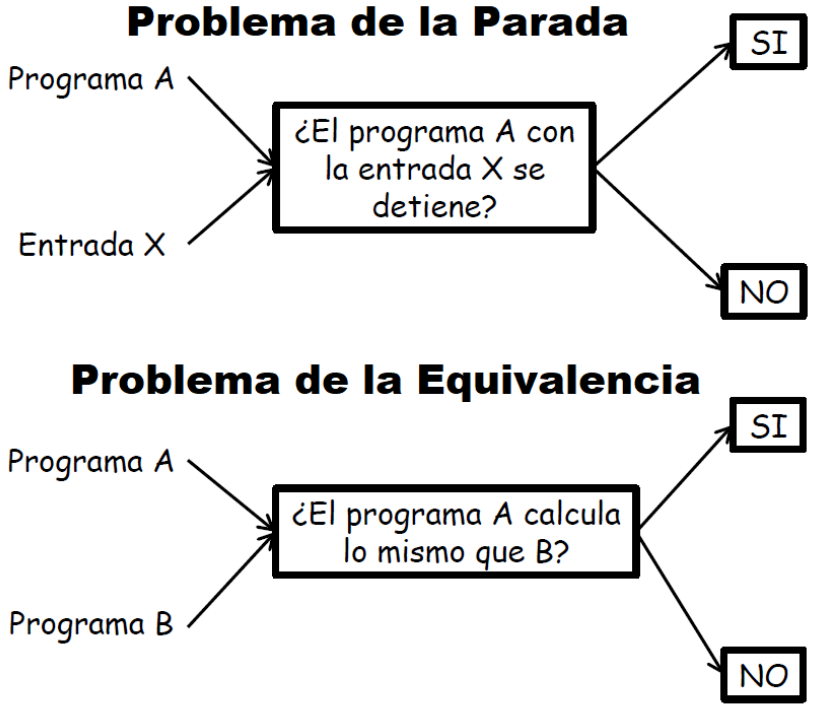
Aquellos que no se pueden resolver mediante un algoritmo, es decir no existe MT que lo pueda resolver.

Problemas Decidibles e Indecidibles



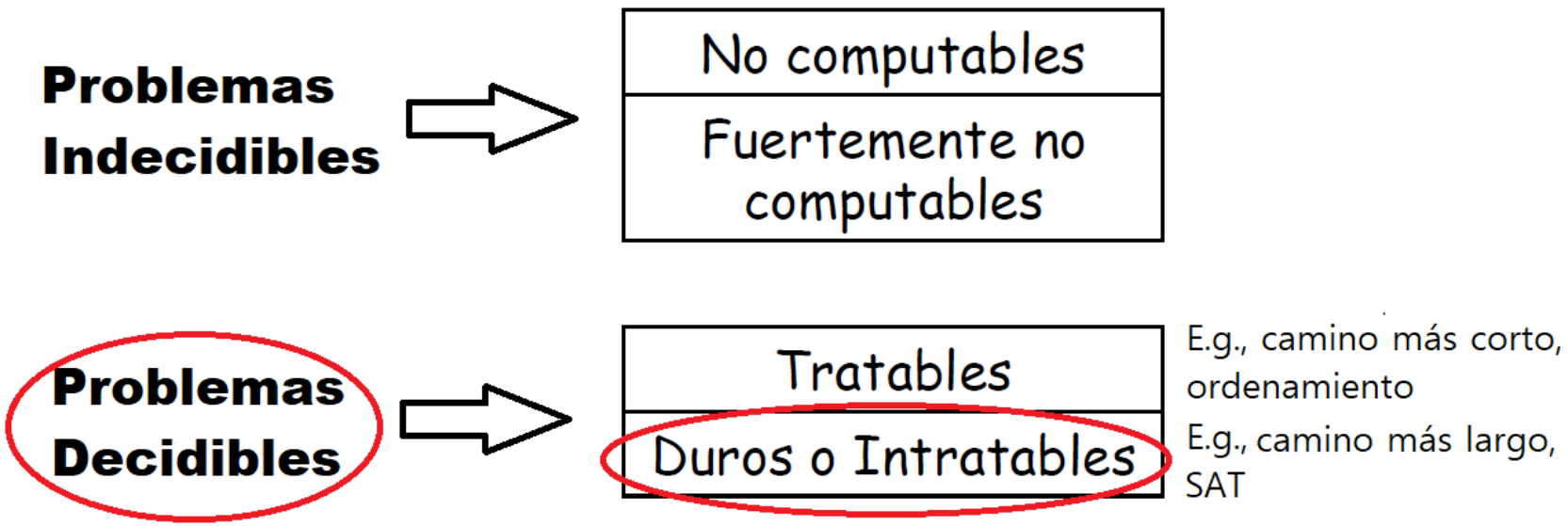
Problemas Indecidibles

Los problemas indecidibles están por fuera del alcance de este curso. Entre ellos se tiene el problema de la parada que es indecidible no computable y el problema de la verificación que es fuertemente no computable:



Problemas Decidibles

Los problemas decidibles están dentro del alcance de este curso y son aquellos para los cuales se puede construir un algoritmo que lleve a una respuesta correcta.



Problemas Decidibles

Los problemas tratables son aquellos que se pueden solucionar con algoritmos polinómiales $O(n^k)$. Esto quiere decir, que si las entradas tienen tamaño n, el peor caso del tiempo de ejecución de estos algoritmos es $O(n^k)$ para alguna constante k.

Los problemas intratables son aquellos para los que no conocemos algoritmos eficientes, es decir, los algoritmos que existen solucionan el problema en tiempo no polinomial $\Omega(a^n)$.

Problemas Decidibles

Tratable: Camino más corto

Dado un grafo $G = (V, E)$, dos nodos $a, b \in V$ y donde cada arista tiene un peso, encontrar el camino más corto entre a y b. Las aristas pueden incluso tener peso negativo y el grafo puede ser dirigido o no dirigido. Este problema se puede resolver en tiempo $O(|V| * |E|)$.

Intratable: Camino más largo

Dado un grafo $G = (V, E)$, dos nodos $a, b \in V$ donde cada arista tiene un peso, encontrar el camino más largo sin ciclos, entre a y b. Determinar el camino simple más largo es un problema intratable (NP-Completo), aun si todos los pesos de las aristas es 1.

Problemas Decidibles

Tratable: Problemas Tour de Euler

Un Tour de Euler de un grafo conexo, dirigido, $G = (V, E)$ es un ciclo que recorre cada arista de G exactamente una vez, aunque puede visitar un vértice más de una vez. Se puede determinar cuándo un grafo tiene un Tour de Euler en tiempo $O(|E|)$ y, de hecho, se pueden encontrar los vértices del Tour en tiempo $O(|E|)$.

Intratable: Ciclo Hamiltoniano

Un Ciclo Hamiltoniano de un grafo dirigido, $G = (V, E)$ es un ciclo simple que contiene cada vértice de V. Determinar si un grafo tiene un ciclo Hamiltoniano es NP-Completo.

¿Cómo saber si un problema es tratable o intratable?¿Cómo los clasifico?

Conceptos Previos

Problemas de Optimización

Problemas donde cada solución factible tiene un valor asociado y se espera encontrar la solución con el mejor valor (i.e., minimizar/maximizar una función objetivo). Por ejemplo, el problema del agente viajero (TSP).

Problemas de Decisión

Problemas cuya respuesta es simplemente SI o NO (i.e., 1 o 0). El estudio de la complejidad aplica directamente sobre problemas de decisión puesto que es conveniente al realizar reducciones. Sin embargo, los problemas de optimización pueden ser expresados fácilmente como problemas de decisión. (p.ej., para una instancia del TSP, ¿existe un tour de tamaño 5 o menos?)

Conceptos Previos

Entre los distintos tipos de máquinas de Turing tenemos las máquinas finitas que se pueden dividir a su vez en:

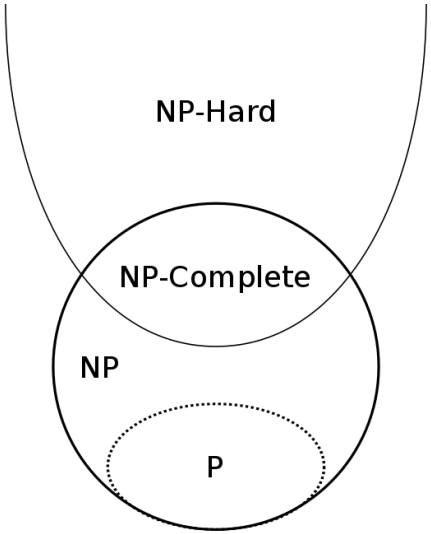
Deterministas

Esta ejecuta un solo paso ante la lectura de un símbolo de entrada (non branching behaviour). Este es el modelo computacional actual.

No deterministas

Ejecuta la mejor acción posible ante la lectura de un símbolo de entrada. Ante una posible ramificación puede explorar todas las bifurcaciones a la vez.

Clasificación de Problemas



La Clase P

La clase P consiste de aquellos problemas que son solucionables en tiempo polinomial por una máquina de turing determinista. Existen muchos problemas que pertenecen a la clase P (envolvente convexa, ordenamiento, búsqueda binaria, camino más corto, . . .).

Clasificación de Problemas

La Clase NP

(NP = Non deterministic polynomial), la clase NP consiste de aquellos problemas que son solucionables en tiempo polinomial por una máquina de turing no determinista. Existen muchos problemas que pertenecen a la clase NP, empezando por todos los de la clase P ($P \subseteq NP$) (problemas de lógica, como SAT; problemas de grafos, como el cubrimiento de vértices; problemas de números, como la partición entera; problemas de programación, como programación entera; . . .).

La Clase NP - versión alternativa

Como las máquinas no deterministas no existen en la realidad, aún, hay una alternativa para reconocer problemas en NP. La clase NP es aquella cuyos problemas son verificables en tiempo polinomial.
Lo que se quiere decir es que dada una solución y un certificado de tamaño polinomial, entonces, es posible verificar, en tiempo polinomial respecto al tamaño de la entrada, que la solución es correcta.

Clasificación de Problemas

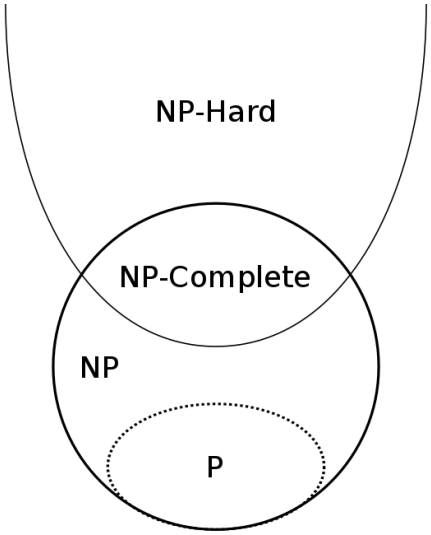
La Clase NP

(NP = Non deterministic polynomial), la clase NP consiste de aquellos problemas que son solucionables en tiempo polinomial por una máquina de turing no determinista. Existen muchos problemas que pertenecen a la clase NP, empezando por todos los de la clase P ($P \subseteq NP$) (problemas de lógica, como SAT; problemas de grafos, como el cubrimiento de vértices; problemas de números, como la partición entera; problemas de programación, como programación entera; . . .).

La Clase NP - versión alternativa

Como las máquinas no deterministas no existen en la realidad, aún, hay una alternativa para reconocer problemas en NP. La clase NP es aquella cuyos problemas son verificables en tiempo polinomial.
Lo que se quiere decir es que dada una solución y un certificado de tamaño polinomial, entonces, es posible verificar, en tiempo polinomial respecto al tamaño de la entrada, que la solución es correcta.

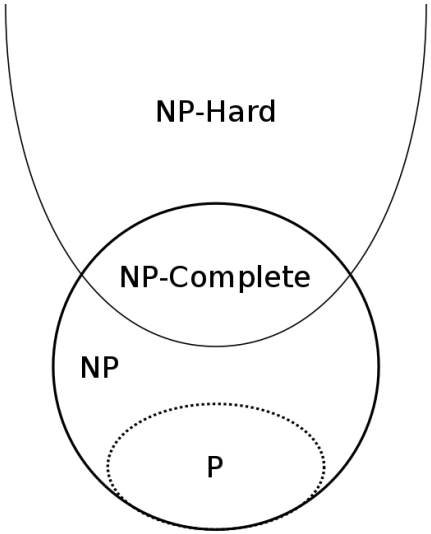
Clasificación de Problemas



La Clase NP

Por ejemplo, para el problema de ciclo hamiltoniano, dado una grafo $G = (V, E)$, una solución sería una secuencia $(v_1, v_2, v_3, ..., v_{|V|})$ de $|V|$ vértices. Verificar que esta secuencia es un ciclo hamiltoniano es muy fácil (polinomial). ¿por qué?

Clasificación de Problemas

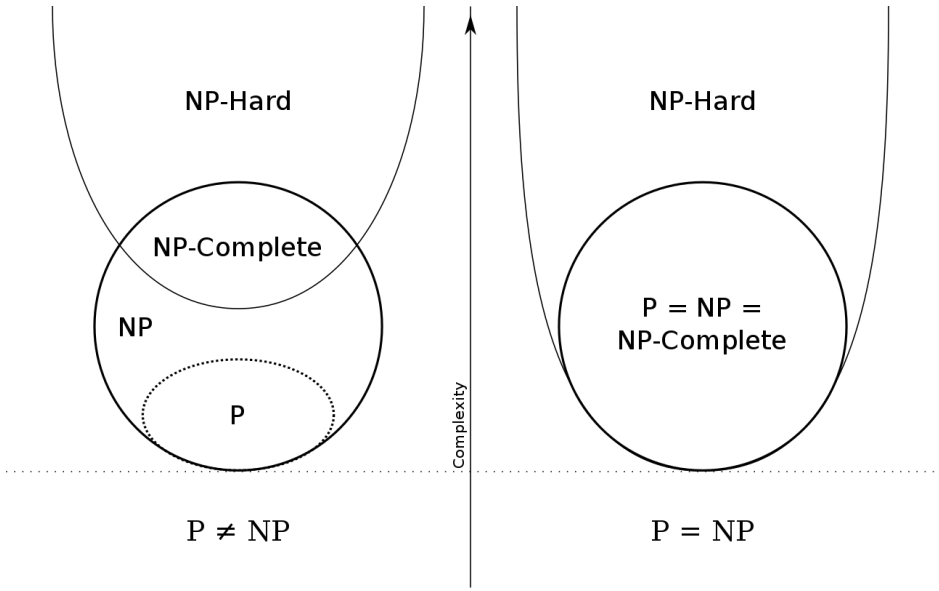


Es fácil notar que $P \subseteq NP$ (¿Por qué?). La pregunta abierta, y que lleva años y años sin ser respondida es:

$$P = NP$$

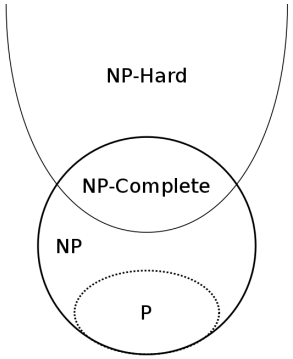
.

Clasificación de Problemas



Hay dos posibles respuestas, cada una con consecuencias distintas (ver gráfica)

Clasificación de Problemas

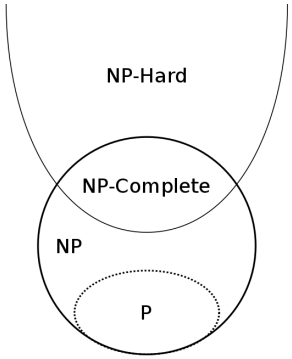


La Clase NPC (intuición)

Son aquellos problemas que:

- **NO** han podido ser solucionados en tiempo polinomial por una máquina determinista
- **SI** pueden ser solucionados en tiempo polinomial por una máquina no determinista
- **Todo problema en NPC** puede ser solucionado a través de ellos.

Clasificación de Problemas

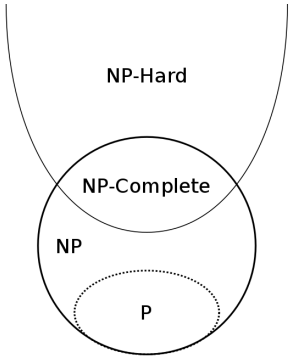


La Clase NPC (intuición)

Son aquellos problemas que:

- **NO** han podido ser solucionados en tiempo polinomial por una máquina determinista
- **SI** pueden ser solucionados en tiempo polinomial por una máquina no determinista
- **Todo problema en NPC** puede ser solucionado a través de ellos.

Clasificación de Problemas

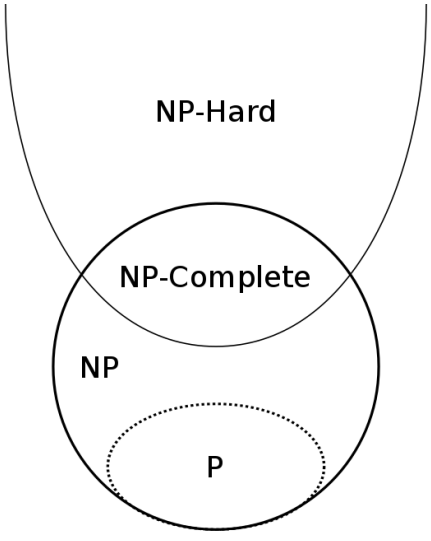


La Clase NPC (intuición)

Son aquellos problemas que:

- **NO** han podido ser solucionados en tiempo polinomial por una máquina determinista
- **SI** pueden ser solucionados en tiempo polinomial por una máquina no determinista
- **Todo problema en NPC** puede ser solucionado a través de ellos.

Clasificación de Problemas

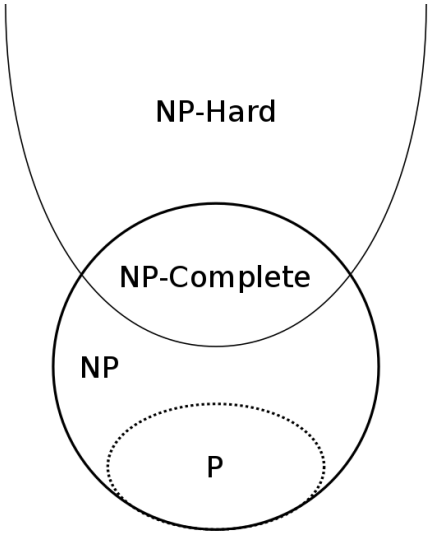


La Clase NPC (un poco más formal, pero no suficiente)

La clase NPC (NP-Completos) es aquella cuyos problemas:

- están en NP
- son tan difíciles (o fáciles) como cualquier problema NPC (¿Cómo decir eso formalmente?)

Clasificación de Problemas



La Clase NPC: su atractivo

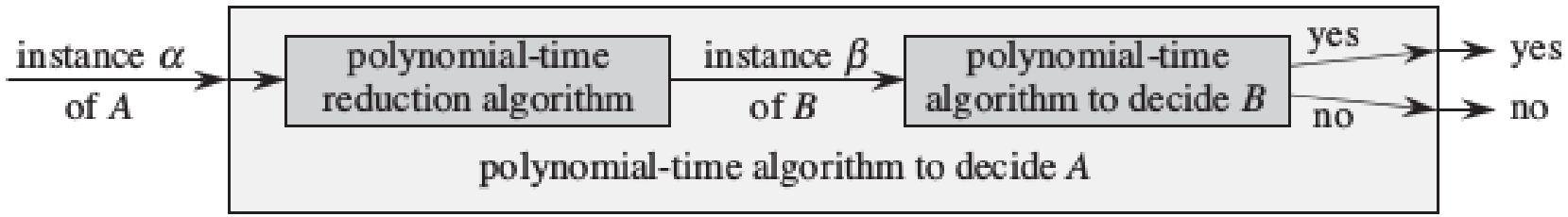
Si un problema NPC puede ser solucionado en tiempo polinomial, entonces todos los problemas NPC pueden también ser resueltos en tiempo polinomial.

Problemas NP-Completos y Reducciones

Cuando nos enfrentamos a producir una solución algorítmica para un problema, bien vale la pena saber de antemano si el problema dado es NP-Completo o no. Por este motivo, debemos tener en cuenta lo siguiente:

- La NP-Completitud no aplica directamente a problemas de optimización. En cambio se aplica a problemas de decisión.
- Es posible convertir un problema de optimización en un problema de decisión imponiendo un límite en el valor que se desea optimizar.

Problemas NP-Completos y Reducciones

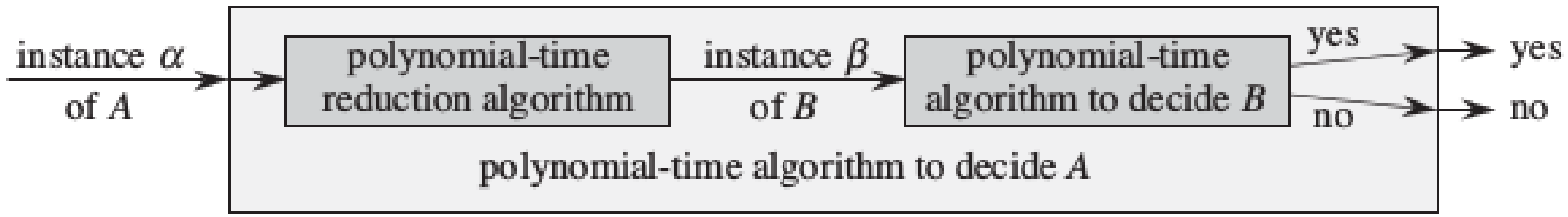


Para mostrar que un problema es tan difícil (duro) como otro se utilizará el concepto de reducción. La figura ilustra el proceso de reducción de un problema A en un problema B.

Una reducción es un procedimiento que transforma en tiempo polinomial cualquier instancia α de un problema A en una instancia β del problema B, y que permite dar la solución al problema A de forma correcta.

Nótese que si tenemos un algoritmo para resolver B en tiempo polinomial, usando la reducción tendremos un algoritmo para resolver A en tiempo polinomial

Problemas NP-Completos y Reducciones

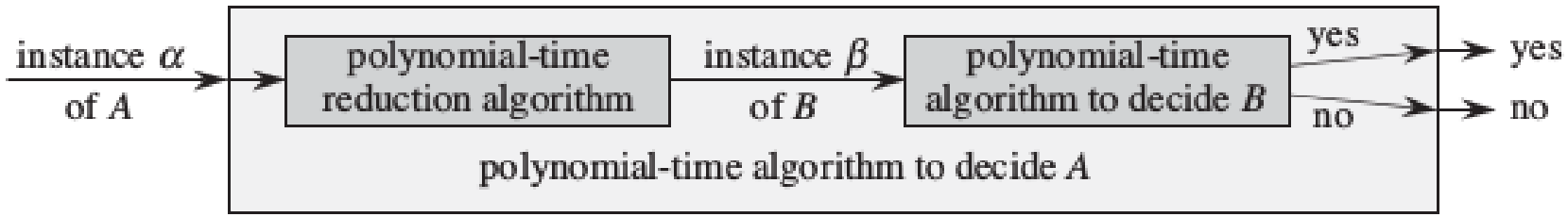


Para mostrar que un problema es tan difícil (duro) como otro se utilizará el concepto de reducción. La figura ilustra el proceso de reducción de un problema A en un problema B.

Una reducción es un procedimiento que transforma en tiempo polinomial cualquier instancia α de un problema A en una instancia β del problema B, y que permite dar la solución al problema A de forma correcta.

Nótese que si tenemos un algoritmo para resolver B en tiempo polinomial, usando la reducción tendremos un algoritmo para resolver A en tiempo polinomial

Problemas NP-Completos y Reducciones

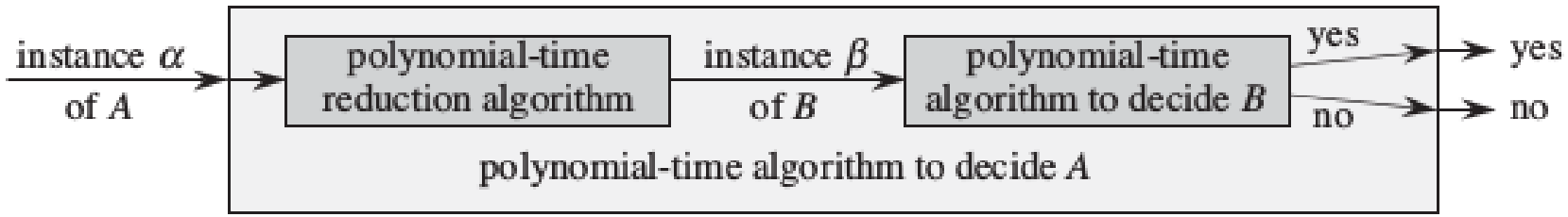


Para mostrar que un problema es tan difícil (duro) como otro se utilizará el concepto de reducción. La figura ilustra el proceso de reducción de un problema A en un problema B.

Una reducción es un procedimiento que transforma en tiempo polinomial cualquier instancia α de un problema A en una instancia β del problema B, y que permite dar la solución al problema A de forma correcta.

Nótese que si tenemos un algoritmo para resolver B en tiempo polinomial, usando la reducción tendremos un algoritmo para resolver A en tiempo polinomial

Problemas NP-Completos y Reducciones

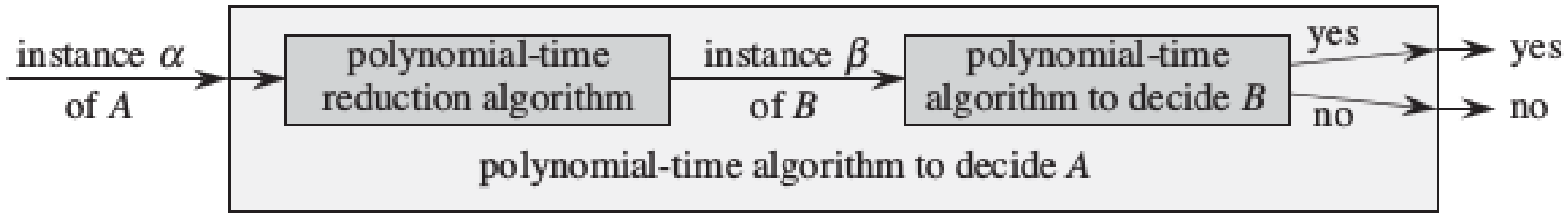


Requerimiento

Una reducción es correcta si:

- Se realiza en tiempo polinomial
- Instancias positivas de A resultan en instancias positivas de B
- Instancias negativas de A resultan en instancias negativas de B

Problemas NP-Completos y Reducciones



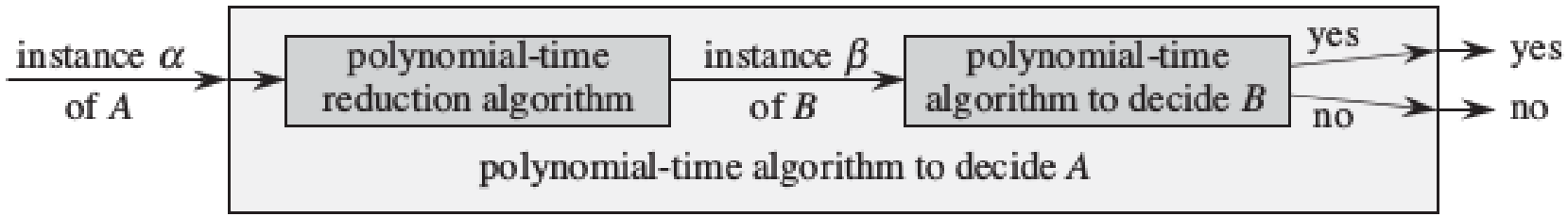
Notación

Se escribe:

$$A \preceq_p B$$

Significa: A reduce a B en tiempo polinomial.
Es decir, si tengo un algoritmo que resuelve B, tengo uno para resolver A.
O sea B es tan o más fuerte/duro/difícil que A

Problemas NP-Completos y Reducciones



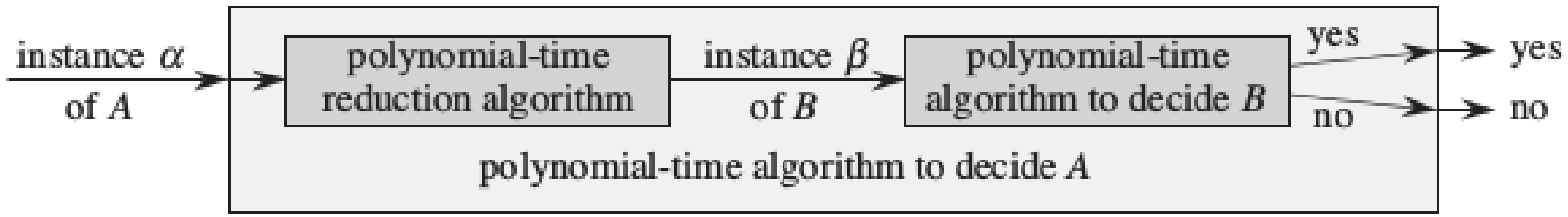
La clase NPC revisitada (1)

Un problema B es NP-Completo ($B \in NPC$) si:

1. (Ser NP) $B \in NP$ (que significa que se verifica rápidamente)
2. (Ser NP duro) $\forall A \in NP : A \preceq_p B$

Es fácil probar (1), pero es difícil probar(2)

Problemas NP-Completos y Reducciones



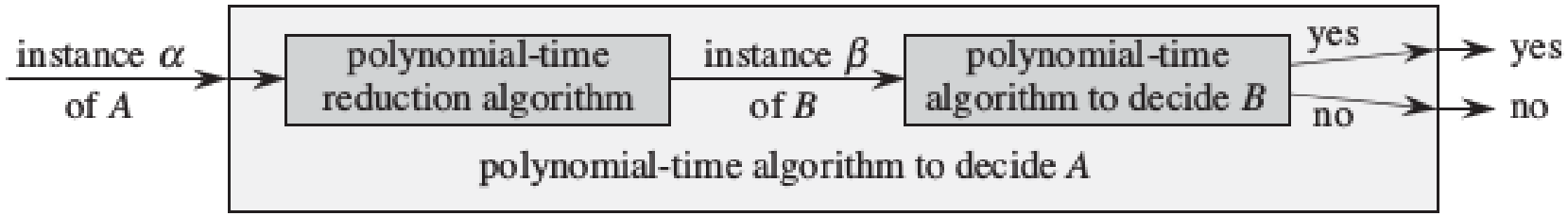
La clase NPC revisitada (1)

Un problema B es NP-Completo ($B \in NPC$) si:

1. (Ser NP) $B \in NP$ (que significa que se verifica rápidamente)
2. (Ser NP duro) $\forall A \in NP : A \preceq_p B$

Es fácil probar (1), pero es difícil probar(2)

Problemas NP-Completos y Reducciones



La clase NPC revisitada (2)

Suponga que ya se sabe que $C \in NPC$.

Teorema: Un problema B es NP-Completo ($B \in NPC$) si:

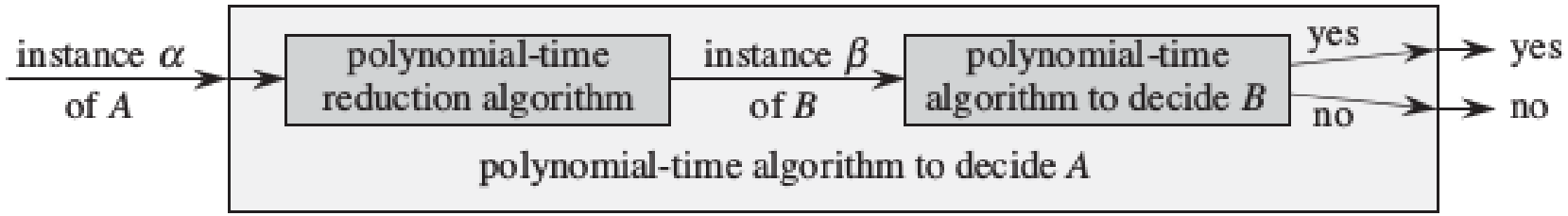
1. (Ser NP) $B \in NP$ (que significa que se verifica rápidamente)
2. (Ser NP duro) $C \preceq_p B$ (en lugar de $\forall A \in NP : A \preceq_p B$)

¿Por qué es cierto el teorema?

Es fácil probar (1), y es fácil probar(2)

Lo difícil es tener un C con esa propiedad

Problemas NP-Completos y Reducciones



La clase NPC revisitada (2)

Suponga que ya se sabe que $C \in NPC$.

Teorema: Un problema B es NP-Completo ($B \in NPC$) si:

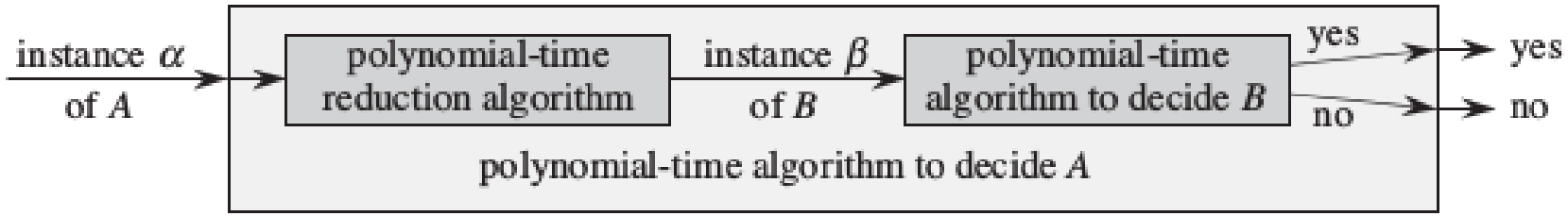
1. (Ser NP) $B \in NP$ (que significa que se verifica rápidamente)
2. (Ser NP duro) $C \preceq_p B$ (en lugar de $\forall A \in NP : A \preceq_p B$)

¿Por qué es cierto el teorema?

Es fácil probar (1), y es fácil probar(2)

Lo difícil es tener un C con esa propiedad

Problemas NP-Completos y Reducciones



La clase NPC revisitada (2)

Suponga que ya se sabe que $C \in NPC$.

Teorema: Un problema B es NP-Completo ($B \in NPC$) si:

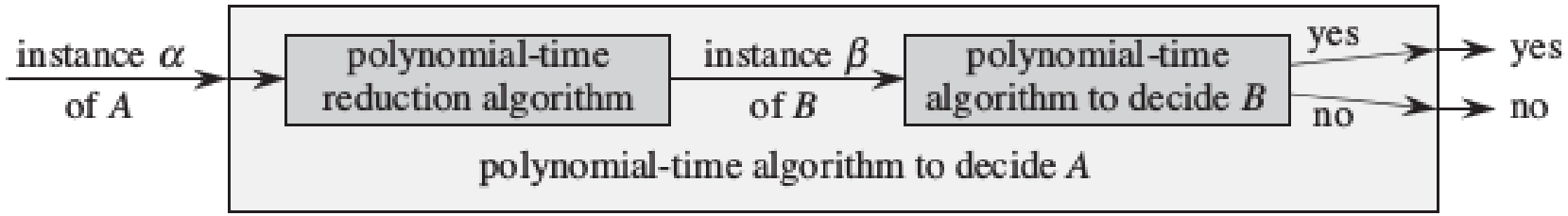
1. (Ser NP) $B \in NP$ (que significa que se verifica rápidamente)
2. (Ser NP duro) $C \preceq_p B$ (en lugar de $\forall A \in NP : A \preceq_p B$)

¿Por qué es cierto el teorema?

Es fácil probar (1), y es fácil probar(2)

Lo difícil es tener un C con esa propiedad

Problemas NP-Completos y Reducciones



La clase NPC revisitada (2)

Suponga que ya se sabe que $C \in NPC$.

Teorema: Un problema B es NP-Completo ($B \in NPC$) si:

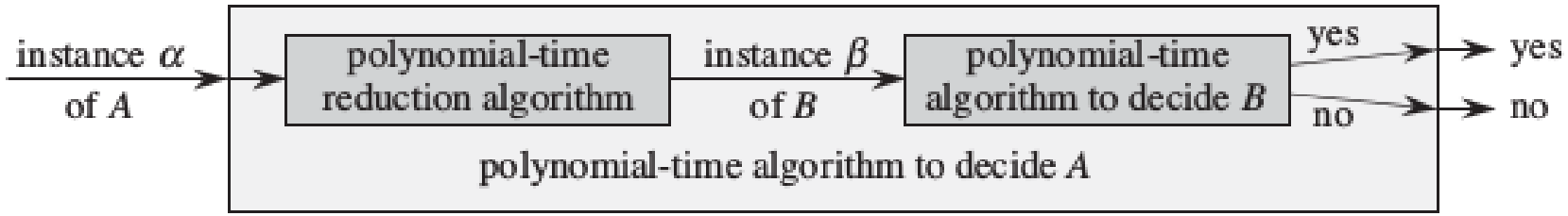
1. (Ser NP) $B \in NP$ (que significa que se verifica rápidamente)
2. (Ser NP duro) $C \preceq_p B$ (en lugar de $\forall A \in NP : A \preceq_p B$)

¿Por qué es cierto el teorema?

Es fácil probar (1), y es fácil probar(2)

Lo difícil es tener un C con esa propiedad

Problema de Satisfactibilidad (SAT)



Generalidades

- Usar la técnica de reducción para probar NP-Completitud, parte de conocer un problema que esté en la clase NPC.
- Para probar que otro problema es NP-Completo es necesario un primer problema NP-Completo.
- Para esto, utilizaremos el problema de satisfactibilidad (SAT), el cual fue demostrado que es NP-Completo, por Cook. (Demostración fuera del alcance de este curso)

Problema de Satisfactibilidad (SAT)

SAT

El problema de satisfactibilidad booleana es el primer problema demostrado NP completo. La demostración fue realizada por Stephen Cook en 1971.

SAT es NP.
Y cualquier problema NPC puede ser reducido a SAT en tiempo polinomial.

Problema de Satisfactibilidad (SAT)

SAT

El problema consiste en un conjunto V de n variables booleanas $v_1, v_2, v_3 ... v_n$ y un conjunto C de m clausulas $c_1, c_2, c_3 ... c_m$ en forma normal conjuntiva (FNC). Se busca si existen valores de las variables que hagan que la expresión sea verdadera (es decir, satisfactible).

Ejemplo de instancia SAT

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

Problema de Satisfactibilidad (SAT)

(Ser NP) SAT es NP

- Si la entrada es satisfactible, es porque existe al menos una asignación de valor a las variables que hacen que la fórmula sea cierta. Dada esa asignación, es muy fácil verificar (reemplazando en cada cláusula) si cada cláusula evalúa a verdadero.
- Si la entrada es insatisfactible, ninguna asignación de variables que me entreguen servirá para verificar que es una instancia positiva.

Problema de Satisfactibilidad (SAT)

(Ser NP duro) SAT es NP-duro (intuición)

- Dada una máquina de Turing no determinista cualquiera que resuelva un problema NPC, se produce una fórmula en FNC de SAT tal que (1) si la máquina aceptaba la entrada, la fórmula es satisfactible, y (2) si la máquina no aceptaba la entrada, la fórmula es insatisfactible.
- El número de variables necesitadas para codificar la máquina en una fórmula es polinomial con respecto al tamaño de la máquina.
- El número de cláusulas de la fórmula es polinomial en el tamaño de la máquina.

Problema de Satisfactibilidad (SAT)

(Ser NP duro) SAT es NP-duro (intuición)

- Dada una máquina de Turing no determinista cualquiera que resuelva un problema NPC, se produce una fórmula en FNC de SAT tal que (1) si la máquina aceptaba la entrada, la fórmula es satisfactible, y (2) si la máquina no aceptaba la entrada, la fórmula es insatisfactible.
- El número de variables necesitadas para codificar la máquina en una fórmula es polinomial con respecto al tamaño de la máquina.
- El número de cláusulas de la fórmula es polinomial en el tamaño de la máquina.

Problema de Satisfactibilidad (SAT)

(Ser NP duro) SAT es NP-duro (intuición)

- Dada una máquina de Turing no determinista cualquiera que resuelva un problema NPC, se produce una fórmula en FNC de SAT tal que (1) si la máquina aceptaba la entrada, la fórmula es satisfactible, y (2) si la máquina no aceptaba la entrada, la fórmula es insatisfactible.
- El número de variables necesitadas para codificar la máquina en una fórmula es polinomial con respecto al tamaño de la máquina.
- El número de cláusulas de la fórmula es polinomial en el tamaño de la máquina.

Problema de Satisfactibilidad (SAT)

SAT es NP-Completo - Implicaciones

- La demostración asume el poder computacional que tienen las máquinas no deterministas.
- Si se llegara a demostrar que SAT puede ser resuelto en tiempo polinomial, entonces cualquier problema NPC puede ser resuelto en tiempo polinomial.
- Al reducir SAT a cualquier problema NP, se demuestra que ese problema es NP-duro.

Lo que sigue

Aumentar nuestro acervo de problemas NP completos

$$SAT \prec_p 3 - SAT \prec_p IP$$

$$3 - SAT \prec_p VC \prec_p MaxClique$$

$$VC \prec_p SubSetSum$$

Fin de la Presentación

¿Preguntas?